

2. Manejo de SQLAlchemy y Bases de Datos en Flask

Inicio



Manejando SQLAlchemy en Flask



1. Introducción a SQLAlchemy



Contenido

Flask-SQLAlchemy es una extensión de Flask que facilita la integración con SQLAlchemy, uno de los ORM (Object Relational Mapping) más poderosos de Python. SQLAlchemy permite interactuar con bases de datos de manera sencilla, mapeando tablas a clases Python y proporcionando un enfoque declarativo para manejar los datos.

Características Principales de Flask-SQLAlchemy

- **Integración Sencilla:** Proporciona una interfaz para conectar Flask con bases de datos compatibles con SQLAlchemy.
- **ORM Declarativo:** Te permite definir tus modelos como clases Python que representan tablas de la base de datos.
- **Soporte Multi-BD:** Compatible con varios sistemas de bases de datos, como SQLite, PostgreSQL, MySQL, entre otros.
- **Consultas Simplificadas:** Incluye herramientas para realizar consultas complejas con una sintaxis Pythonica.

Configuración Básica

Instalación: Para usar Flask-SQLAlchemy, primero debes instalarlo:

```
pip install flask-sqlalchemy
```

Configuración Inicial: Configurar Flask-SQLAlchemy requiere especificar la URL de conexión a la base de datos.

Ejemplo:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///mi_base_de_datos.db"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False # Desactiva las s
db = SQLAlchemy(app)
```



Definición de Modelos

Los modelos en SQLAlchemy representan las tablas en la base de datos. Los atributos de una clase corresponden a las columnas de la tabla.

Ejemplo:

```
class Usuario(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    edad = db.Column(db.Integer)

    def __repr__(self):
        return f"<Usuario {self.nombre}>"
```

Operaciones CRUD

Una vez definidos los modelos, puedes realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar).

1. Crear

```
nuevo_usuario = Usuario(nombre="Juan", email="juan@example.com", edad=3
db.session.add(nuevo_usuario)
db.session.commit()
```



2. Leer

```
# Obtener todos los usuarios
usuarios = Usuario.query.all()

# Obtener un usuario por ID
usuario = Usuario.query.get(1)

# Filtrar usuarios
usuarios_mayores = Usuario.query.filter(Usuario.edad > 18).all()
```

3. Actualizar

```
usuario = Usuario.query.get(1)
usuario.nombre = "Juan Pérez"
db.session.commit()
```

4. Eliminar

```
usuario = Usuario.query.get(1)
db.session.delete(usuario)
db.session.commit()
```

Relaciones entre Tablas

Flask-SQLAlchemy soporta relaciones entre tablas utilizando claves foráneas y asociaciones.

Ejemplo: Relación Uno a Muchos

```
class Publicacion(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    titulo = db.Column(db.String(100), nullable=False)
    contenido = db.Column(db.Text, nullable=False)
    usuario_id = db.Column(db.Integer, db.ForeignKey("usuario.id"), nul
    usuario = db.relationship("Usuario", backref="publicaciones")
```



Migraciones con Flask-Migrate

Para manejar cambios en los modelos de manera estructurada, se utiliza la extensión **Flask-Migrate**, basada en Alembic.

Instalación

```
pip install flask-migrate
```

Configuración

```
from flask_migrate import Migrate
migrate = Migrate(app, db)
```

Comandos Comunes

Iniciar Migraciones:

```
flask db init
```

Crear una Migración:

```
flask db migrate -m "Descripción del cambio"
```

Aplicar Cambios:

```
flask db upgrade
```

Buenas Prácticas

Dividir Modelos en Módulos: Mantén los modelos en un archivo o paquete separado para un mejor mantenimiento.

Validaciones: Utiliza validaciones tanto en el modelo como en el lado de la base de datos para garantizar la integridad de los datos.

Transacciones: Asegúrate de manejar correctamente las transacciones con `db.session.commit()` y utiliza `rollback` en caso de errores.

Optimización de Consultas: Emplea consultas específicas (`query.filter()`, `query.join()`) para minimizar la carga en la base de datos.

Uso de Migraciones: Siempre utiliza Flask-Migrate para aplicar cambios en los modelos, evitando realizar alteraciones manuales.

2. Configuración de SQLAlchemy en el Proyecto



Contenido

1. Instalando SQLAlchemy

Procedemos a abrir una terminal en VSCode y escribiremos el siguiente comando:

```
pip install -U Flask-SQLAlchemy
```

```
Collecting greenlet!=0.4.17 (from sqlalchemy>=2.0.16->Flask-SQLAlchemy)
  Downloading greenlet-3.1.1-cp311-cp311-win_amd64.whl.metadata (3.9 kB)
Requirement already satisfied: colorama in c:\users\usuario\appdata\local\programs\python\python311\lib\site-packages (from click>=8.1.3->flask>=2.2.5->Flask-SQLAlchemy) (0.4.6)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\usuario\appdata\local\programs\python\python311\lib\site-packages (from Jinja2>=3.1.2->flask>=2.2.5->Flask-SQLAlchemy) (3.0.2)
Downloading flask_sqlalchemy-3.1.1-py3-none-any.whl (25 kB)
  Downloading SQLAlchemy-2.0.36-cp311-cp311-win_amd64.whl (2.1 MB)
    2.1/2.1 MB 7.0 MB/s eta 0:00:00
  Downloading greenlet-3.1.1-cp311-cp311-win_amd64.whl (298 kB)
    298.9/298.9 KB 9.0 MB/s eta 0:00:00
  Downloading typing_extensions-4.12.2-py3-none-any.whl (37 kB)
Installing collected packages: typing-extensions, greenlet, sqlalchemy, Flask-SQLAlchemy
Successfully installed Flask-SQLAlchemy-3.1.1 greenlet-3.1.1 sqlalchemy-2.0.36 typing-extensions-4.12.2
[notice] A new release of pip is available: 24.0 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\WebApp\Projects\todo-list>
```

Abrimos el archivo `__init__.py` y agregamos la importación de SQLAlchemy y la cadena de conexión.

Importamos SQLAlchemy y creamos la extensión:

```
todor > 🐍 __init__.py > ⚙️ create_app
1   # Importando la clase Flask
2   from flask import Flask, render_template
3   from flask_sqlalchemy import SQLAlchemy
4
5   # Creando extensión de la base de datos
6   db = SQLAlchemy()
7
8   # Creando función de control
9   def create_app():
10      |
```

Creamos la configuración y la inicialización:

```
11  # Creando la variable de iniciación
12  app = Flask(__name__)
13
14  # Configuración del proyecto
15  app.config.from_mapping(
16      |     DEBUG = True,
17      |     SECRET_KEY = 'dev esit',
18      |     SQLALCHEMY_DATABASE_URI = "sqlite:///todolist.db"
19  )
20
21  # Iniciando conexión
22  db.init_app(app)
23
24  # Registrando Blueprint
25  from . import todo
26  app.register_blueprint(todo.bp)
27
```

3. Creando función de migración a la base datos

Agregamos la final una función que permita que una vez definidos todos los modelos y tablas, llama a `SQLAlchemy.create_all()` para crear el esquema de tabla en la base de datos. Esto requiere un contexto de aplicación, es decir, se creará el contenido que no exista:

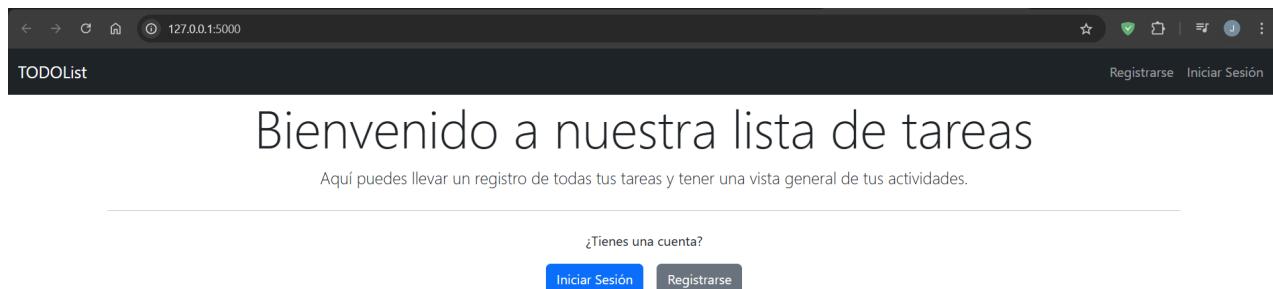
```
30
31     # Definiendo rutas
32     @app.route('/')
33     def index():
34         return render_template('index.html')
35
36     # Creación de tablas a partir de los modelos
37     with app.app_context():
38         db.create_all()
39
40     return app
```

4. Revisando la ejecución del proyecto y creación de la base de datos

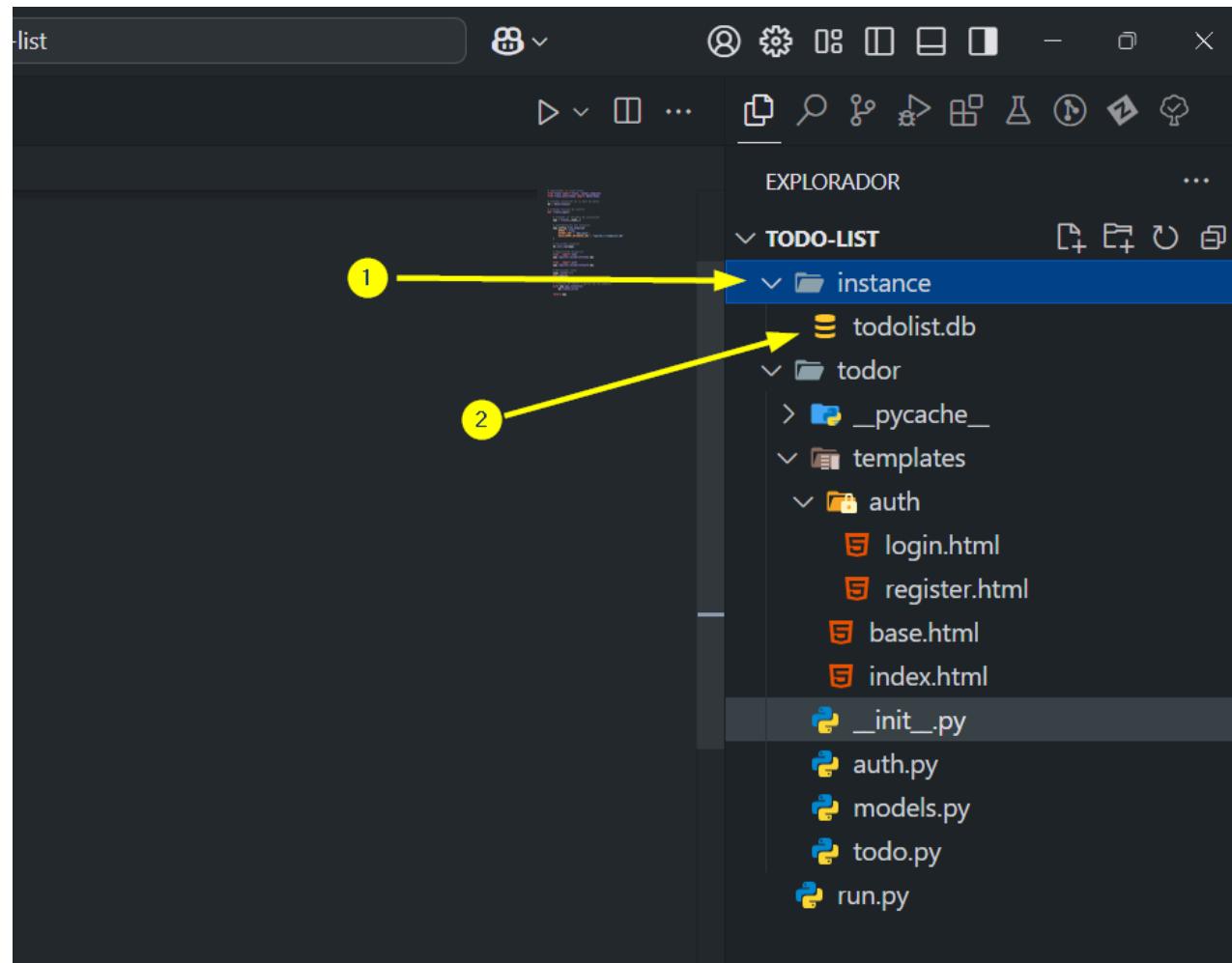
Con el comando `python run.py`, iniciamos el servidor:

```
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 pro
.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 929-180-456
|
```

Revisamos el navegador:



Revisamos que se haya creado un directorio llamado instance, donde estará nuestra base de datos todolist.db:



3. Creando Modelos del Proyecto

Creación de Modelos en Flask



Contenido

1. Instalando Extensión SQLite para VSCode

Dentro de las extensiones para Visual Studio Code, procedemos a instalar SQLite Viewer, esta nos ayudará a visualizar el contenido de nuestras bases de datos y tablas creadas en SQLite:

The screenshot shows the VS Code Marketplace interface. At the top, there's a search bar with the text "Extensión: SQLite Viewer". Below it, the "SQLite Viewer" extension card is displayed, featuring a blue icon of a database with a feather, the name "SQLite Viewer", the developer "Florian Klampfer", the version "1.540.181", and a rating of "★★★★★ (58)". Below the card, there are buttons for "Deshabilitar" (Disable), "Desinstalar" (Uninstall), and "Actualización automática" (Automatic update). At the bottom of the card, there are tabs for "DETALLES", "CARACTERÍSTICAS", and "REGISTRO DE CAMBIOS".

SQLite Viewer for VS Code

A quick and easy SQLite viewer for VS Code, inspired by DB Browser for SQLite and Airtable.

The main view shows a file explorer on the left with various SQLite databases and files listed. On the right, there's a preview pane showing a table from the "northwind.db" database. The preview pane has columns for "Last Name", "First Name", "Title", "Title of Resposability", "PhotoPath", and "Photo". It lists 9 rows of data, each corresponding to an employee in the database. To the right of the preview pane, there's a "Marketplace" sidebar with details about the extension, including its identifier, version, publication date, and last update. There are also sections for "Categorías" (Categories) and "Recursos" (Resources).

Abrimos el archivo models.py y crearemos la clase para el modelo de usuario:

```
# Importando la base de datos
from todor import db

# Creando clase para usuario
class User(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    username = db.Column(db.String(20), unique = True, nullable = False)
    password = db.Column(db.Text, nullable = False)

    def __init__(self, username, password):
        self.username = username
        self.password = password

    def __repr__(self):
        return f'<User: {self.username}>'
```



Explicando el código

Atributos de la Clase

- **id**: Es una clave primaria (`primary_key=True`) que es ideal para identificar de forma única a cada usuario.
- **username**: Tiene restricciones de unicidad (`unique=True`) y no puede ser nulo (`nullable=False`), lo cual es adecuado para nombres de usuario únicos.
- **password**: Almacena contraseñas. Sin embargo, es importante encriptarlas antes de guardarlas.

Constructor: El método `__init__` facilita la creación de instancias del modelo.

Representación (`__repr__`): Proporciona una representación útil para depuración al mostrar el nombre del usuario.

3. Creando el modelo para todo

Abrimos el archivo models.py y crearemos la clase para el modelo de todo:

```
# Creando clase para todo
class Todo(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    created_by = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    title = db.Column(db.String(100), nullable = False)
    desc = db.Column(db.Text)
    state = db.Column(db.Boolean, default = False)

    def __init__(self, created_by, title, desc, state = False):
        self.created_by = created_by
        self.title = title
        self.desc = desc
        self.state = state

    def __repr__(self):
        return f'<Todo: {self.title} >'
```



Explicando el código

Atributos Principales:

- id: Es la clave primaria que identifica de forma única cada tarea.
- created_by: Es una clave foránea que relaciona la tarea con un usuario del modelo User. Esto permite manejar tareas asociadas a usuarios específicos.
- title: Representa el título de la tarea y es obligatorio (nullable=False).
- desc: Almacena una descripción más detallada de la tarea.
- state: Es un booleano que indica si la tarea está completada, con un valor predeterminado de False.

Constructor: Permite inicializar el modelo con los valores necesarios.

Representación (`__repr__`): Hace que las instancias del modelo sean fáciles de identificar en la depuración.

4. Migración y Flask Shell



SECRETARÍA DE
INNOVACIÓN



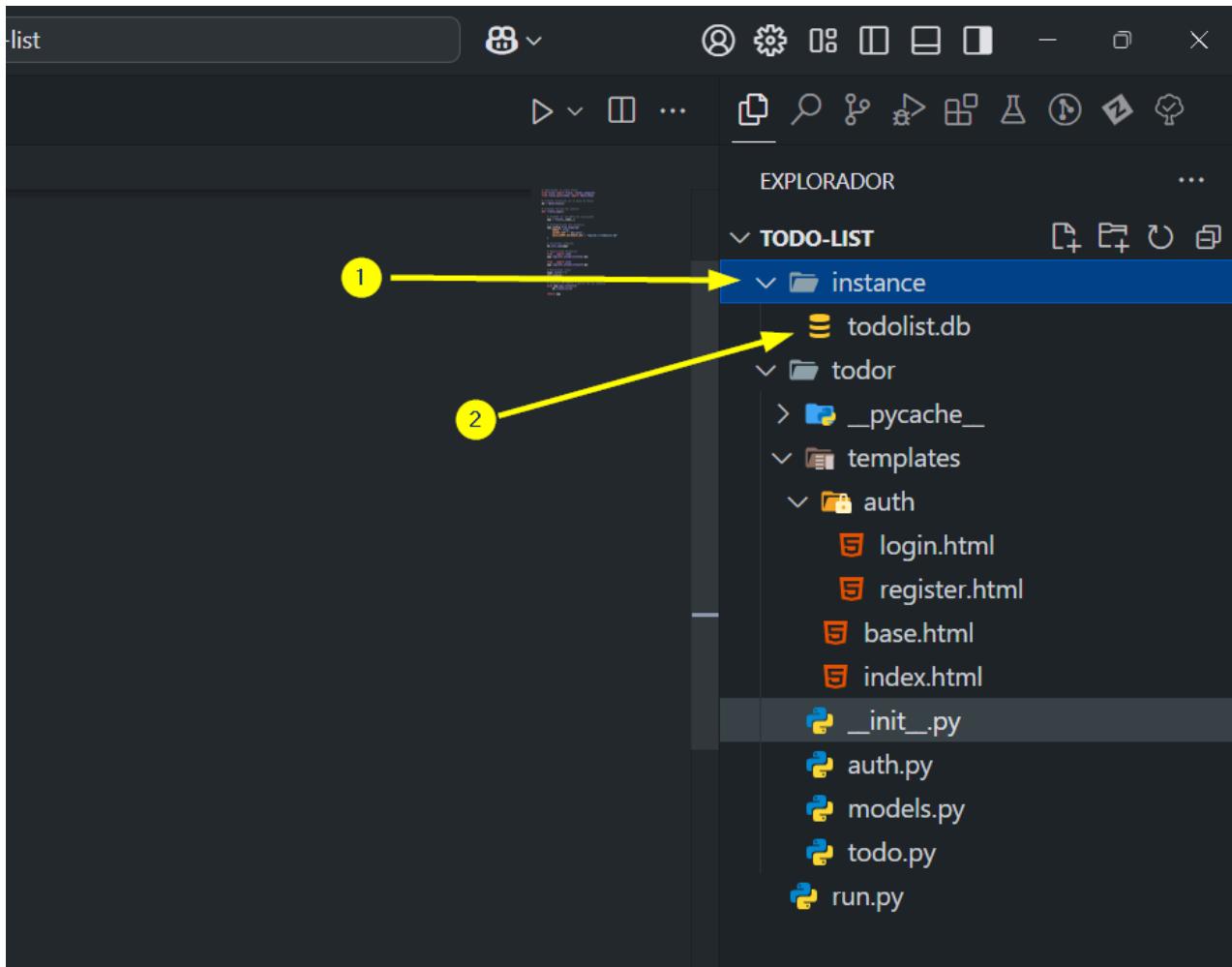
Migración de Modelos y Flask Shell



Contenido

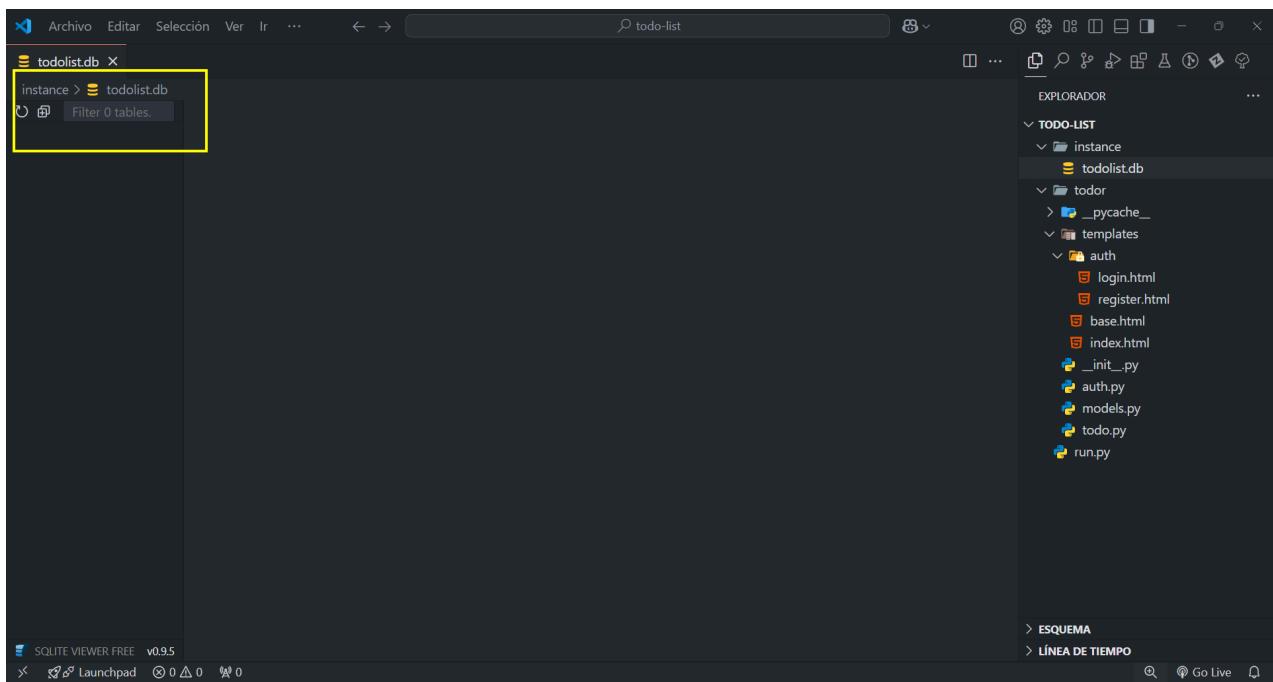
1. Verificando la base de datos

Verificamos que la base de datos se encuentre en instance:



2. Usando la extensión de SQLite

Si damos doble clic en el archivo todolist.db, veremos lo siguiente:



Notemos que nos dice Files: 0 tables, es porque no tenemos tablas dentro de la base de datos, vamos a proceder a utilizar la migración de tipo OCR para crear las tablas a partir del modelo.

3. Creando la Migración de los Modelos

Abrimos el archivo auth.py:

```
todor > 🐍 auth.py > ...
1 # Importando Blueprint
2 from flask import Blueprint, render_template
3 from . import models
4
5 # Creando instancia
6 bp = Blueprint('auth', __name__, url_prefix='/auth')
7
8 #Creado ruta y función
9 @bp.route('/register')
```

Si ejecutamos el comando python run.py para iniciar el servidores veremos:

```
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 pro
.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 929-180-456
```

Si volvemos darle doble clic en todolist.db, veremos las tablas creadas:

instance > 📁 todolist.db		Rows: 0						Filter 0 rows...
Filter 2 tables.								
▼ TABLES		id	create...	title	desc	state		
>	todo							
>	user							
		1						

instance > todolist.db

Filter 2 tables. Rows: 0 Filter 0 rows...

TABLES

> todo
> user

	id	username	password
1			

5. Registrando Usuarios



SECRETARÍA DE
INNOVACIÓN



Registrando Datos



Contenido

Registrando Usuarios: Importaciones

Abrimos el archivo auth.py, y realizamos las siguientes adiciones de importaciones:

```
auth.py M X
todor > auth.py > login
You, hace 2 minutos | 1 author (You)
# Importando funcionalidades de la app
from flask import (
    Blueprint, render_template, request, url_for, redirect, flash
)
from werkzeug.security import generate_password_hash, check_password_hash
from .models import User
from todor import db

# 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')      You, hace 2 semanas • proyecto webapp con flask
```

Registrando Usuarios: Manejo de Datos

Ubicamos el apartado @bp.route('/register') y realizamos la siguiente modificación:

```

#Creado ruta y función
@bp.route('/register', methods = ('GET', 'POST'))
def register():
    #Validación de datos
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # Encriptando password
        user = User(username, generate_password_hash(password))

        # Consultado a la base de datos
        user_name = User.query.filter_by(username = username).first()
        if user_name == None:
            db.session.add(user)
            db.session.commit()
            return redirect(url_for('auth.login'))

    return render_template('auth/register.html')

```

Registrando Usuarios: Explicando código

Decorador @bp.route('/register'): Este decorador vincula la función register a la ruta /register de la aplicación. Cuando un usuario accede a esta URL, la función correspondiente se ejecuta. bp es un Blueprint, que ayuda a organizar las rutas y funcionalidades de la aplicación en módulos.

Definición de la función register(): Define la lógica que se ejecutará al acceder a la ruta /register. Dentro de esta función se manejarán las solicitudes del cliente, como mostrar el formulario o procesar los datos enviados.

Validación del método HTTP: Comprueba si la solicitud enviada por el cliente es de tipo POST, lo que indica que el formulario fue enviado con datos. Si no es POST, la función ignora este bloque y pasa al final.

Obtención de datos del formulario: Extrae los valores enviados en el formulario, específicamente el nombre de usuario y la contraseña. Estos valores se obtienen de los campos del formulario HTML mediante request.form.

Encriptación de la contraseña: Crea un objeto User (probablemente un modelo de base de datos) utilizando el nombre de usuario proporcionado y la contraseña

encriptada. Esto asegura que la contraseña nunca se almacene en texto plano.

Consulta en la base de datos: Busca en la base de datos si ya existe un usuario con el mismo nombre de usuario. Si encuentra un registro coincidente, devuelve ese objeto; de lo contrario, devuelve None.

Comprobación y registro del usuario: Si el nombre de usuario no está registrado:

- Añade el nuevo usuario al sistema, almacenándolo en la base de datos.
- Confirma el cambio con un commit para que los datos se guarden de forma permanente.
- Redirige al usuario a la página de inicio de sesión.

Renderización de la plantilla: Si la solicitud no es de tipo POST, o después de procesar una solicitud que no cumple con las condiciones, se devuelve el formulario de registro para que el usuario pueda ingresar sus datos.

Registrando Usuarios: Revisión

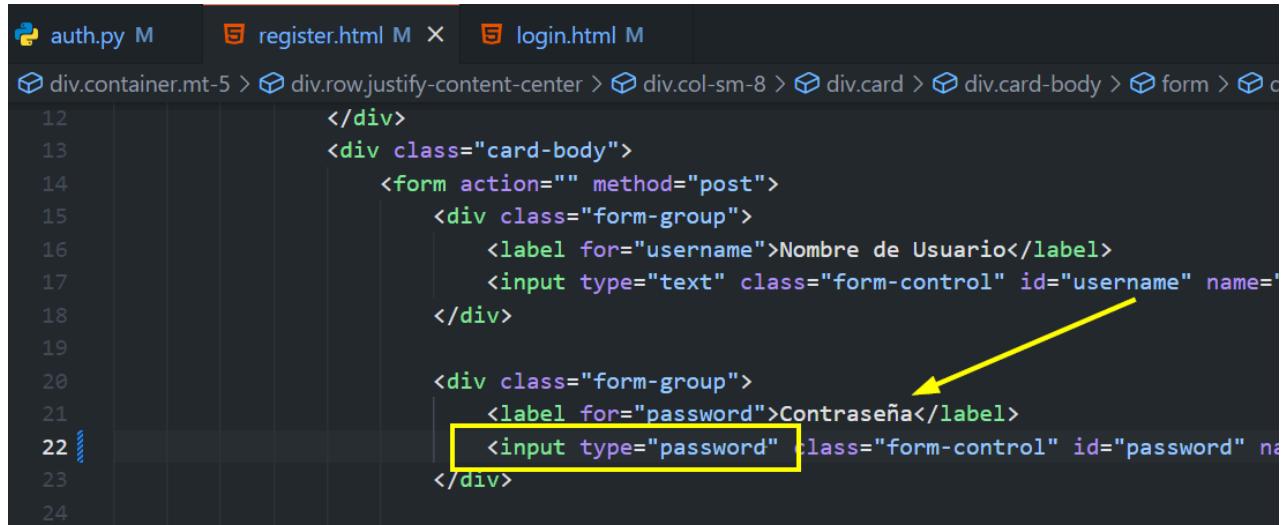
El código anterior debe quedar de la siguiente manera:

```
todor > 🐍 auth.py > ⚙ register
 9  # Creando instancia
10 bp = Blueprint('auth', __name__, url_prefix='/auth')
11
12 #Creado ruta y función
13 @bp.route('/register', methods = ('GET', 'POST'))
14 def register():
15     #Validación de datos
16     if request.method == 'POST':
17         username = request.form['username']
18         password = request.form['password']
19
20         # Encriptando password
21         user = User(username, generate_password_hash(password))
22
23         # Consultado a la base de datos
24         user_name = User.query.filter_by(username = username).first()
25         if user_name == None:
26             db.session.add(user)
27             db.session.commit()      You, hace 24 segundos • Uncommitted changes
28             return redirect(url_for('auth.login'))
29
30         return render_template('auth/register.html')
31
32 @bp.route('/login')
33 def login():
34     return render_template('auth/login.html')
```

Registrando Usuarios: Modificando HTML

vamos al archivo register.html y el archivo login.html y cambiamos el type de text a password:

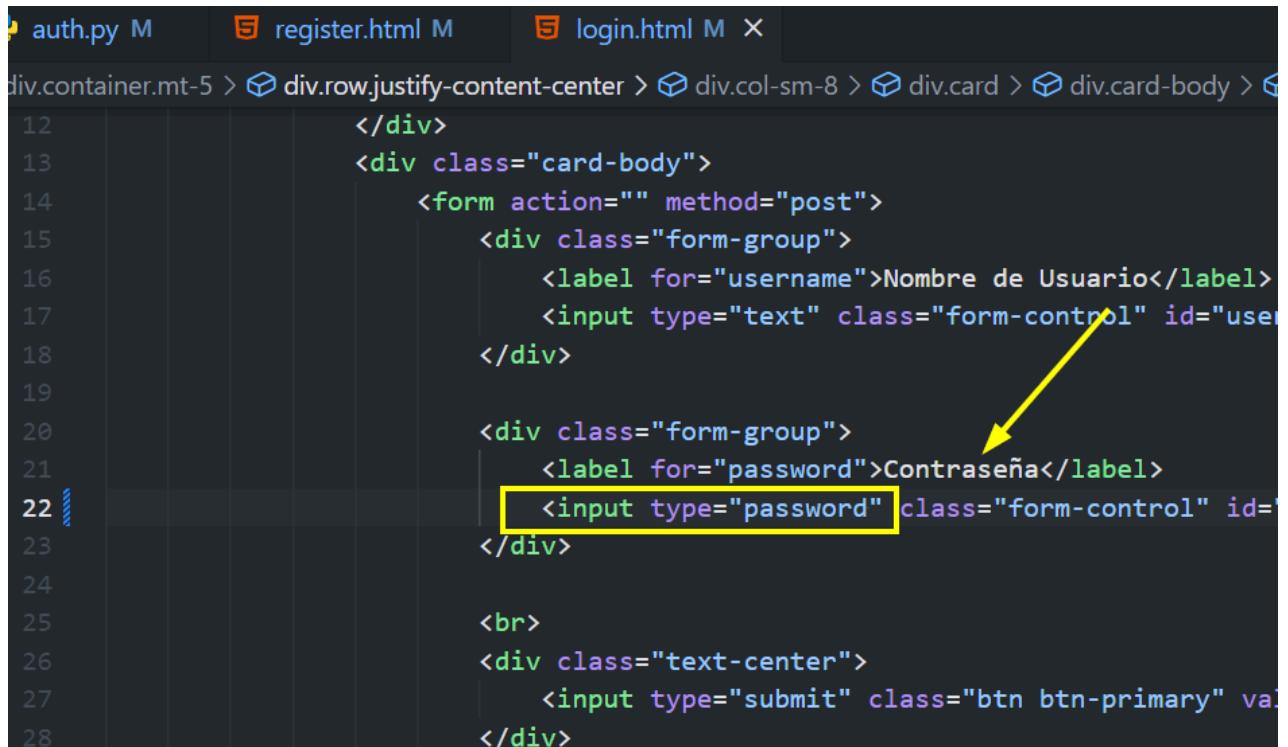
register.html



```
auth.py M register.html M login.html M

```

login.html



```
auth.py M register.html M login.html M
div.container.mt-5 > div.row.justify-content-center > div.col-sm-8 > div.card > div.card-body > form > div
12         </div>
13     <div class="card-body">
14         <form action="" method="post">
15             <div class="form-group">
16                 <label for="username">Nombre de Usuario</label>
17                 <input type="text" class="form-control" id="username" name="username" required="required" value="<input type="text" class="form-control" id="username" name="username" required="required" value=">">
18             </div>
19
20             <div class="form-group">
21                 <label for="password">Contraseña</label>
22                 <input type="password" class="form-control" id="password" name="password" required="required" value="<input type="password" class="form-control" id="password" name="password" required="required" value=">">
23             </div>
24
25             <br>
26             <div class="text-center">
27                 <input type="submit" class="btn btn-primary" value="Iniciar Sesión" style="width: 150px; height: 40px; font-size: 16px; border-radius: 5px; border: none; background-color: #007bff; color: white; padding: 10px; margin-bottom: 10px;">
28             </div>

```

Registrando Usuarios: Prueba de Registro

Iniciamos el servidor en el caso se encuentre inactivo, vamos al navegador y realizamos una prueba de registro de usuario:

127.0.0.1:5000/auth/register

TODOLIST

Registrarse Iniciar Sesión

Registrar Usuario

Nombre de Usuario
mate032

Contraseña

Registrar Usuario

Al dar clic, redirigirá a login:

127.0.0.1:5000/auth/login

TODOLIST

Registrarse Iniciar Sesión

Iniciar Usuario

Nombre de Usuario

Contraseña

Iniciar Sesión

Si abrimos la base de datos con la extensión dentro de VSCode:

auth.py M todolist.db M

instance > todolist.db

Filter 2 tables.

TABLES

- > todo
- > user

Rows: 1

	id	username	password
+	1	mate032	script:32768:8:1\$tXiDsxsjEbOM918C\$c0c05ef25b96743f...
	2		

Filter 1 rows... Upgrade

6. Manejo de Errores



Manejo de Errores



Contenido

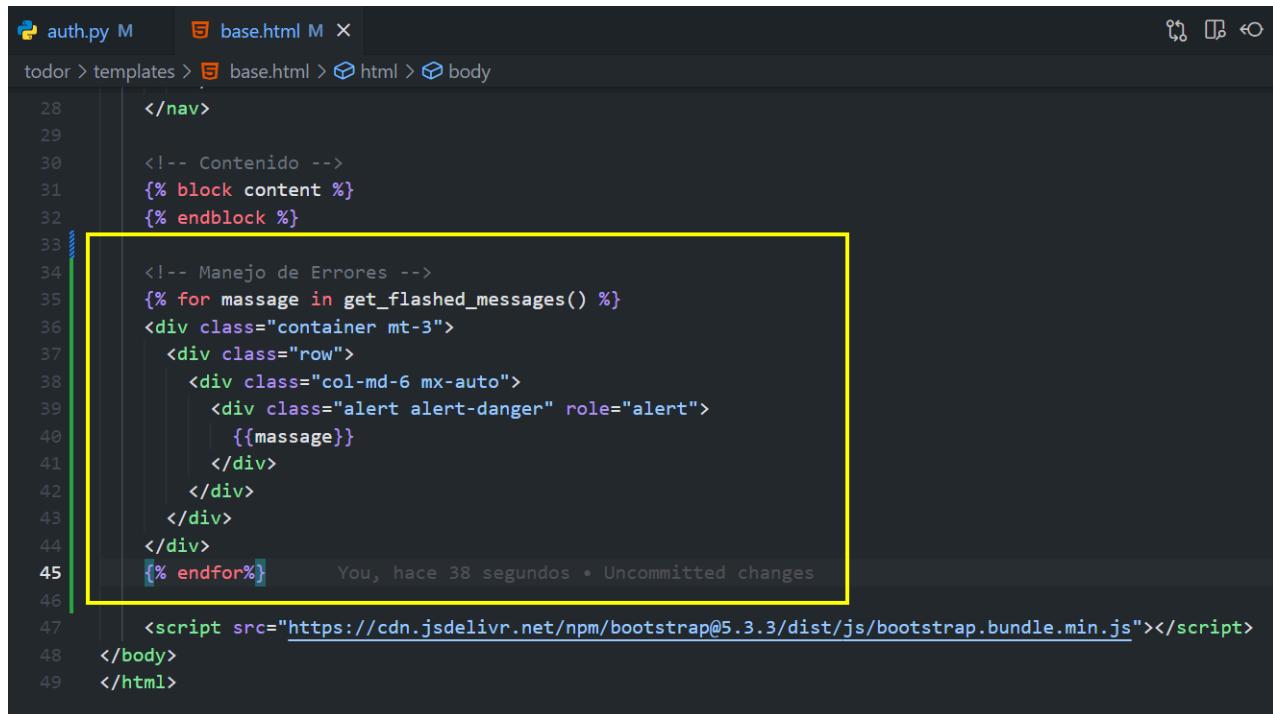
Manejo de Errores: Modificando Registro de Usuario

Abrimos el archivo auth.py, ubicamos el bloque de registro y agregamos lo siguiente:

```
todor > python auth.py > register
11
12     #Creado ruta y función
13     @bp.route('/register', methods = ('GET', 'POST'))
14     def register():
15         #Validación de datos      You, hace 1 segundo • Uncommitted changes
16         if request.method == 'POST':
17             username = request.form['username']
18             password = request.form['password']
19
20             # Encriptando password
21             user = User(username, generate_password_hash(password))
22
23             # Manejo de errores
24             error = None
25
26             # Consultando a la base de datos
27             user_name = User.query.filter_by(username = username).first()
28             if user_name == None:
29                 db.session.add(user)
30                 db.session.commit()
31                 return redirect(url_for('auth.login'))
32             else:
33                 error = f"El usuario {username} ya se encuentra registrado"
34                 flash(error)
35
36             return render_template('auth/register.html')
37
38     @bp.route('/login')
```

Abrimos base.html y agregamos lo siguiente:

```
<!-- Manejo de Errores -->
{% for message in get_flashed_messages() %}
<div class="container mt-3">
    <div class="row">
        <div class="col-md-6 mx-auto">
            <div class="alert alert-danger" role="alert">
                {{message}}
            </div>
        </div>
    </div>
</div>
{% endfor%}
```



The screenshot shows a code editor with two tabs: auth.py and base.html. The base.html tab is active, displaying the following Jinja template code:

```
</nav>

{% block content %}
{% endblock %}


{% for message in get_flashed_messages() %}
<div class="container mt-3">
    <div class="row">
        <div class="col-md-6 mx-auto">
            <div class="alert alert-danger" role="alert">
                {{message}}
            </div>
        </div>
    </div>
</div>
{% endfor%}

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

A yellow rectangular box highlights the error handling code (lines 34-45). A status bar at the bottom right indicates "You, hace 38 segundos • Uncommitted changes".

Manejo de Errores: Explicando el Código

Línea 1: Comentario `<!-- Manejo de Errores -->` Es un comentario en HTML que describe el propósito del bloque de código: gestionar y mostrar mensajes flash.

Línea 2: Bucle Jinja `{% for message in get_flashed_messages() %}`, Inicia un bucle que recorre todos los mensajes flash almacenados en la sesión. La función `get_flashed_messages()` de Flask se utiliza para obtener una lista de los mensajes

almacenados temporalmente. Cada mensaje en esa lista se procesa en el bloque del bucle.

Líneas 3-9: Estructura de HTML para mostrar los mensajes, Se utiliza una estructura de Bootstrap para mostrar cada mensaje dentro de un contenedor estilizado. Vamos a desglosarla:

Línea 3: `<div class="container mt-3">`, Crea un contenedor con márgenes en la parte superior (mt-3), utilizando las clases de Bootstrap.

Línea 4: `<div class="row">`, Define una fila para organizar los elementos dentro del contenedor.

Línea 5: `<div class="col-md-6 mx-auto">`, Crea una columna de tamaño mediano (6 unidades de las 12 disponibles en el sistema de rejilla de Bootstrap). La clase mx-auto centra horizontalmente esta columna dentro de la fila.

Línea 6-8: Alerta con el mensaje, Se usa una clase de alerta de Bootstrap (alert alert-danger) para mostrar el mensaje. alert-danger: Estiliza la alerta con un fondo rojo, indicando que se trata de un mensaje de error o advertencia. role="alert": Atributo de accesibilidad que indica que este elemento es una alerta para usuarios que utilicen tecnologías de asistencia.

Dentro del `<div>` se inserta el mensaje usando la sintaxis de Jinja2 (`{{message}}`), lo que inyecta el contenido dinámico.

Línea 10: Fin del bucle `{% endfor %}`, Cierra el bloque del bucle Jinja2. Este bloque asegura que se rendericen todas las alertas correspondientes a los mensajes flash obtenidos.

Flujo de Ejecución

- Flask almacena mensajes temporales (flash) en la sesión del usuario usando `flask.flash()`.
- Cuando se llama a `get_flashed_messages()`, Flask entrega todos los mensajes pendientes y los elimina de la sesión.
- Este código recorre esos mensajes y los muestra en la página web, uno por uno, dentro de una alerta estilizada.

Manejo de Errores: Realizando la Prueba

Vamos a registrar el mismo usuario de la prueba anterior:

The screenshot shows a web browser window with the URL `127.0.0.1:5000/auth/register`. The page title is "TODOLIST". In the top right corner, there are links for "Registrarse" and "Iniciar Sesión". The main content is a form titled "Registrar Usuario". It has two input fields: "Nombre de Usuario" and "Contraseña", both of which are currently empty. Below the fields is a blue button labeled "Registrar Usuario". At the bottom of the form, there is a pink rectangular box containing the text "El usuario mate032 ya se encuentra registrado".

Registrar Usuario

Nombre de Usuario

Contraseña

Registrar Usuario

El usuario mate032 ya se encuentra registrado

Notarás que muestra el error indicando que ya esta registrado, validando la existencia de un único usuario por persona.
