Figure 1: Logo

Documentación de Formulario Web: Programacion 4

Jose Enrique Tejada Ramirez

December 4, 2024

Contents

1 Instrucciones

Desarrollar una API con FastAPI: Crear un servidor web que reciba datos desde un cliente HTTP a través de un formulario web.

Conectar una Base de Datos: Usar una base de datos relacional para almacenar los datos enviados por el usuario.

Validación de Datos: Implementar validaciones para asegurar que los datos recibidos son correctos y cumplen con los requisitos establecidos.

Documentación Automática de la API: Utilizar las funcionalidades de documentación automática que ofrece FastAPI mediante OpenAPI.

Uso de ORM (SQLAlchemy): Implementar un ORM para interactuar con la base de datos y manejar el almacenamiento de los datos de forma eficiente.

2 Requisitos

El primer capítulo de la documentación, sirve como una introducción al lenguaje de programación Python. Los puntos clave incluyen:

1. FastAPI: Para crear el servidor web y la API RESTful.
2. Base de Datos: Puedes elegir entre SQLite, MySQL o PostgreSQL. Para simplicidad y facilidad de uso, se recomienda SQLite.
3. ORM: Usar SQLAlchemy para gestionar la base de datos.
4. Formulario en HTML: Crear una página HTML para que el usuario ingrese sus datos a través de un formulario.
5. Validación: Implementar validaciones de los datos recibidos, por ejemplo, verificar que el correo electrónico tenga un formato válido y que los campos como la edad sean números válidos.
6. Documentación Automática: Utilizar la funcionalidad que FastAPI ofrece para la documentación automática de la API.

3 Tecnologias Usadas

3.1 Python:

1. Utilizado como lenguaje principal en el backend.
2. Framework FastAPI para crear la API y manejar las solicitudes.
3. Uvicorn: Permite que la aplicación FastAPI sea accesible como un servidor web.
4. Pydantic: Lo uso en los esquemas para validar y serializar datos (en el archivo schemas.py
5. SQLAlchemy para manejar la interacción con la base de datos.
6. Jinja2 para renderizar plantillas HTML dinámicas.

3.2 HTML:

1. Lenguaje de marcado utilizado para la estructura del formulario y la página web.
2. Renderizado dinámicamente con Jinja2 para mostrar listas de usuarios y mensajes.

3.3 CSS:

1. Utilizado para diseñar y estilizar el frontend.
2. Incluye diseño responsivo, estilos para formularios, botones, y mensajes de error/éxito.

3.4 SQL:

1. Usado indirectamente a través de SQLAlchemy para definir y consultar modelos de datos.

4 Estructura del codigo:

```
MY_FASTAPI_PROJECT/
 templates/
    form.html
 database.py
 main.py
 models.py
 schemas.py
```

5 Documentación del Archivo form.html

Este archivo define la interfaz de usuario para un sistema de gestión de usuarios, permitiendo registrar, listar y eliminar usuarios. A continuación, se explica el propósito y funcionalidad de cada parte del archivo.

5.1 Propósito General

El objetivo principal del archivo es ofrecer una experiencia de usuario intuitiva para la gestión de datos de usuarios. Esto incluye:

1. Registrar nuevos usuarios con un formulario.
2. Mostrar una lista de usuarios existentes.
3. Eliminar usuarios específicos.
4. Presentar mensajes de retroalimentación al usuario a través de un modal emergente.

5.2 Secciones Clave

1. Estructura HTML Básica

La estructura HTML establece el diseño principal:

Encabezado (¡head¿): Configura el idioma, los metadatos y el estilo visual mediante CSS.

Cuerpo (¡body¿): Contiene los elementos visibles, como el formulario, la lista de usuarios y el modal para mensajes.

2. Formulario de Registro html

5.3 templates/form.html - Formulario Web

Listing 1: form.html

```
<form method="post" action="/users/">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>

    <label for="age">Age:</label>
    <input type="number" id="age" name="age" min="0" required>

    <button type="submit">Submit</button>
</form>
```

Qué hace:
Permite al usuario registrar su nombre, correo electrónico y edad.
Envía los datos al servidor mediante un método POST.

Por qué se hace:
Para facilitar la entrada de datos en el sistema.
Añadir nuevos usuarios a la lista gestionada por el sistema.

3. Mensajes de Retroalimentación

5.4 templates/form.html - Formulario Web

Listing 2: form.html

```
{% if message %}
<div id="message" data-message="{{ message }}"></div>
{% endif %}
{% if error %}
<div id="error" data-message="{{ error }}"></div>
{% endif %}
```

Qué hace: Integra mensajes generados por el servidor (éxito o error) en la página.

Por qué se hace: Proveer retroalimentación directa al usuario después de realizar acciones. Informar al usuario si su acción fue exitosa o si ocurrió algún problema (como un correo duplicado).

5.5 Estilos CSS

Listing 3: form.html

```
body {
    font-family: 'Arial', sans-serif;
    background-color: #f4f4f9;
    color: #333;
    padding: 20px;
}
```

Qué hace:
Define la apariencia del formulario, los botones, la lista y el modal.

Por qué se hace:
Para que la interfaz sea visualmente atractiva y fácil de usar.

Mejorar la usabilidad y profesionalismo del sistema.

5.6 Funcionalidad con JavaScript

Listing 4: form.html

```
function showModal(message) {
    const modal = document.getElementById('messageModal');
    const modalMessage = document.getElementById('modalMessage');
    modalMessage.textContent = message;
    modal.style.display = 'flex';
}
function closeModal() {
    const modal = document.getElementById('messageModal');
    modal.style.display = 'none';
}
```

Qué hace:

Abre y cierra el modal cuando hay un mensaje que mostrar.
Por qué se hace:

Hacer la página más dinámica sin necesidad de recargarla.

Enfocar al usuario en la información importante.

5.7 Conclusión

Este archivo combina tecnologías como HTML, CSS, plantillas de servidor y JavaScript para construir una interfaz rica y funcional. Cada elemento cumple un propósito específico, asegurando que los usuarios puedan interactuar con el sistema de manera eficiente y sin fricciones.

6 Documentacion de Database.py

El archivo database.py es el núcleo que conecta la aplicación con la base de datos. Su propósito principal es establecer y configurar la comunicación con una base de datos SQLite utilizando SQLAlchemy, un ORM (Object-Relational Mapping) que permite interactuar con la base de datos de manera más intuitiva y eficiente.

Listing 5: database.py

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

6.1 Propósito General:

El archivo configura la base de datos y prepara las herramientas necesarias para:

1. Conectar con la base de datos de manera segura y eficiente.
2. Definir las tablas y modelos que representan la estructura de datos.
3. Gestionar sesiones para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

6.2 Beneficios de Este Enfoque

Eficiencia: La conexión y las sesiones están configuradas para manejar múltiples operaciones sin necesidad de establecer una nueva conexión cada vez.

Escalabilidad: Este diseño permite cambiar fácilmente a otras bases de datos (como PostgreSQL o MySQL) modificando solo la URL de conexión.

Modularidad: Al separar esta configuración en un archivo independiente, es más fácil mantener y reutilizar el código.

6.3 Conclusión

El archivo database.py es fundamental para la arquitectura del proyecto. Establece las bases para que el resto de la aplicación pueda interactuar con la base de datos de manera limpia y eficiente. Este diseño asegura una separación clara de responsabilidades, permitiendo una mayor flexibilidad y control sobre las operaciones en la base de datos.

7 Documentación del Archivo models.py

El archivo models.py define la estructura y las reglas de la tabla users en la base de datos. Es el lugar donde se utiliza SQLAlchemy para mapear clases Python a tablas en la base de datos, haciendo que la interacción con esta sea más intuitiva y expresiva.

Listing 6: database.py

```
from sqlalchemy import Column, Integer, String, CheckConstraint
from database import Base

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
    age = Column(Integer, nullable=False)
    __table_args__ = (
        CheckConstraint('edad >= 0', name='check_edad_positive'),
    )
```

7.1 Propósito General

Definir el modelo User, que representa una tabla en la base de datos llamada users, con sus columnas, restricciones, y reglas de validación.

7.2 Cómo Funciona Todo Junto

Tabla y Clase: La clase User actúa como un puente entre la aplicación y la tabla users en la base de datos.

Columnas y Propiedades: Cada atributo de la clase representa una columna en la tabla con reglas específicas como unicidad y valores no nulos.

Restricciones: Las reglas como CheckConstraint se aplican automáticamente cuando se inserta o actualiza un dato.

7.3 Beneficios de Este Diseño

Legibilidad: La estructura de la tabla se define de manera clara y comprensible en Python.

Validación Incorporada: Las restricciones aseguran que los datos sean válidos sin necesidad de lógica adicional.

Modularidad: Este archivo puede ser modificado o ampliado fácilmente para agregar más tablas o columnas.

7.4 Conclusión

El archivo models.py es esencial para definir las reglas y la estructura de la base de datos de manera centralizada y orientada a objetos. Esto no solo facilita la interacción con la base de datos, sino que también asegura la integridad y consistencia de los datos almacenados.

8 Documentación del Archivo schemas.py

El archivo schemas.py define las estructuras de datos y las validaciones que se usarán para manejar información de entrada y salida en la API. Con la ayuda de Pydantic, se especifican reglas y restricciones que aseguran la consistencia de los datos antes de interactuar con la base de datos.

Listing 7: Schemas.py

```
from pydantic import BaseModel, EmailStr, Field, validator
from pydantic.networks import EmailStr

class UserCreate(BaseModel):
    name: str  # Usamos 'str' y aplicamos la validaci n manualmente

    email: EmailStr  # Email validado con la clase EmailStr

    age: int = Field(..., ge=0, description="La edad debe ser un n mero positiv
# ge=0 para garantizar que sea >= 0

    @validator('name')
    def validate_name(cls, v):
        # Asegurarse de que solo tenga letras y espacios
        if not v.isalpha() and not all(c.isspace() for c in v):
            raise ValueError('El nombre solo puede contener letras y espacios')
        return v

    class Config:
        # Esto asegura que los datos sean correctamente validados y convertidos
        orm_mode = True
```

8.1 Propósito General

Proporcionar un esquema para validar y estructurar los datos de los usuarios antes de procesarlos. Esto garantiza que la información sea segura, válida y cumpla con las reglas de negocio.

8.2 Cómo Funciona Todo Junto

Creación de Usuario: Cuando se envían datos al API para crear un usuario, UserCreate valida que el nombre, correo y edad cumplan con las reglas especificadas.

Validaciones Automáticas: Pydantic maneja las validaciones básicas (ejemplo: correo válido, edad positiva) mientras que el validador personalizado revisa reglas más complejas como el formato del nombre.

Integración con la Base de Datos: El modo ORM permite que los datos validados se conviertan directamente en instancias de los modelos de la base de datos.

8.3 Conclusión

El archivo schemas.py asegura que los datos enviados a la API sean consistentes y seguros, implementando tanto validaciones básicas como personalizadas. Al trabajar junto con los modelos y la base de datos, forma una capa crítica para mantener la integridad de la aplicación.

9 Resumen del Archivo main.py

El archivo define la API de la aplicación usando FastAPI, con rutas para manejar usuarios en una base de datos SQLite. A continuación, los puntos más importantes:

Listing 8: Main.py

```
from fastapi import FastAPI, Depends, Form, Request, HTTPException
from sqlalchemy.orm import Session
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.templating import Jinja2Templates
import models, schemas
from database import engine, SessionLocal

app = FastAPI()
models.Base.metadata.create_all(bind=engine)

# Configuraci n de Jinja2 para manejar las plantillas
templates = Jinja2Templates(directory="templates")

# Dependencia para obtener la sesi n de la base de datos
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Ruta para eliminar un usuario
@app.post("/users/{user_id}/delete", response_class=HTMLResponse)
def delete_user(user_id: int, db: Session = Depends(get_db)):
    user = db.query(models.User).filter(models.User.id == user_id).first()
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    db.delete(user)
    db.commit()
    # Redirigir a la p gina principal despu s de eliminar
    return RedirectResponse(url="/", status_code=303)

# Lista de usuarios obtenidos de la base de datos
@app.get("/users/", response_class=HTMLResponse)
def get_users(request: Request, db: Session = Depends(get_db)):
    users = db.query(models.User).all()
    return templates.TemplateResponse("form.html", {"request": request, "users":

# Crear un nuevo usuario
```

```
@app.post("/users/", response_class=HTMLResponse)
def create_user(
    request: Request,
    name: str = Form(...),
    email: str = Form(...),
    age: int = Form(...),
    db: Session = Depends(get_db),
):
    # Verificar si el correo ya est  en uso
    existing_user = db.query(models.User).filter(models.User.email == email).fir
    if existing_user:
        users = db.query(models.User).all()  # Obtengo la lista de usuarios
        return templates.TemplateResponse(
            "form.html",
            {
                "request": request,
                "error": "El correo electr nico ya est  en uso, por favor usa
                "users": users,  # Paso la lista de usuarios
            },
        )

    if age < 0:
        users = db.query(models.User).all()  # Obtengo la lista de usuarios
        return templates.TemplateResponse(
            "form.html",
            {"request": request, "error": "Age cannot be negative.", "users": us
        )
    if not name.replace("-", "").isalpha():
        users = db.query(models.User).all()  # Obtengo la lista de usuarios
        return templates.TemplateResponse(
            "form.html",
            {"request": request, "error": "El nombre solo acepta valores de text
        )

    # Crear el nuevo usuario
    user = models.User(name=name, email=email, age=age)
    db.add(user)
    db.commit()
    db.refresh(user)
    users = db.query(models.User).all()  # Obtengo la lista de usuarios
    return templates.TemplateResponse(
        "form.html",
        {
            "request": request,
            "message": "Usuario creado exitosamente!",
            "users": users,  # Paso la lista de usuarios
```

```
        },
    )

# Ruta principal para cargar el formulario
@app.get("/", response_class=HTMLResponse)
def read_form(request: Request, db: Session = Depends(get_db)):
    users = db.query(models.User).all()
    return templates.TemplateResponse("form.html", {"request": request, "users":
```

10 1. Configuración Inicial

Inicialización de FastAPI:
app = FastAPI()

Creación de la Base de Datos:
models.Base.metadata.create$_a ll(bind = engine)$

Plantillas con Jinja2: Configuración para renderizar HTML con form.html ubicado en la carpeta templates.

10.1 Dependencia para la Base de Datos

Se define una función para gestionar la conexión a la base de datos:

Listing 9: Main.py

```
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

10.2 Rutas Principales

a) Ruta para Leer el Formulario Inicial Carga el formulario principal y lista los usuarios:

Listing 10: Main.py

```
@app.get("/", response_class=HTMLResponse)
def read_form(request: Request, db: Session = Depends(get_db)):
    users = db.query(models.User).all()
    return templates.TemplateResponse("form.html", {"request": request, "users":
```

10.3 Conclusión

El archivo implementa un flujo completo de CRUD (crear, leer, y eliminar) para gestionar usuarios, integrando plantillas HTML y validaciones robustas, asegurando una experiencia de usuario clara y funcional.

11 Conclusión General del Proyecto

Este proyecto es una aplicación web simple pero funcional desarrollada con FastAPI, que demuestra las capacidades de un sistema CRUD para gestionar usuarios en una base de datos SQLite. Combina tecnologías modernas y eficientes para lograr una experiencia completa, tanto en backend como en frontend, destacando los siguientes aspectos clave:

Gestión de Datos: La base de datos permite realizar operaciones básicas como creación, lectura y eliminación de registros, asegurando integridad mediante validaciones específicas en los modelos y esquemas.

Validaciones Robustas: Se implementan restricciones en los datos de entrada (nombre, correo electrónico y edad) para garantizar la calidad y consistencia de los datos.

Interfaz Amigable: A través de la integración con Jinja2, se facilita una interfaz web dinámica y clara, que permite interactuar con los datos sin necesidad de herramientas externas.

Arquitectura Modular: La estructura del proyecto (archivos separados para configuración, modelos, esquemas y lógica de la aplicación) favorece la escalabilidad, el mantenimiento y la comprensión del código.

Accesibilidad: Utiliza SQLite como base de datos, lo que simplifica la configuración y hace que el proyecto sea fácilmente portable.

En general, este proyecto es un excelente ejemplo para entender conceptos básicos de desarrollo web, manejo de bases de datos, validación de datos y creación de interfaces dinámicas. Es un punto de partida ideal para proyectos más complejos o para aprender a integrar otras herramientas y tecnologías.