3.56 PROYECTO:

2DA. EVALUACIÓN.

UNIVERSIDAD AUTONOMA DE CHIHUAHUA.
FACULTAD DE INGENERIA.

368027 Sandoval Granados Paloma Ivonne.

Docente: Jose Saul de Lira Miramontes.

BASES DE DATOS AVANZADAS - 7CC2.

10 Noviembre, 2025

Aplicación web de gestión para un blog construido en Python usando el micro-framework Flask.

Arquitectura de la aplicación.

- operacionesCRUD.py:

Maneja la comunicación con la base de datos MongoDB.

- controladores/:

Maneja las rutas (URLs) y la lógica de qué hacer cuando un usuario visita una página.

- templates/:

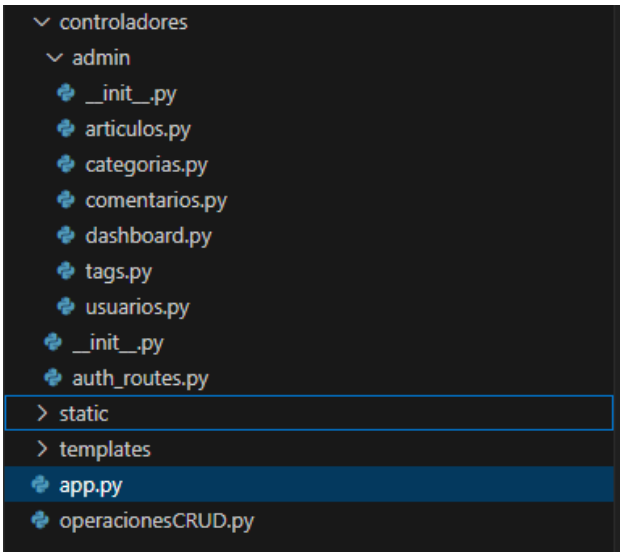
Contiene todos los archivos HTML que el usuario ve en el navegador.

- static/:

Almacena los archivos CSS (estilos) e imágenes.

- app.py:

Es el punto de entrada de la aplicacion.



Base de datos.

La base de datos se encuentra almacenada en Mongo Atlas con el nombre de proyecto "Proyecto2BDII".

Proyecto2BDII

LOGICAL DATA SIZE: 1.57KB STORAGE SIZE: 176KB INDEX SIZE: 176KB TOTAL COLLECTIONS: 5 CREATE COLLECTION

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
articles	1	1.22KB	1.22KB	36KB	1	36KB	36KB
categories	2	80B	40B	36KB	1	36KB	36KB
comments	0	0B	0B	32KB	1	32KB	32KB
tags	4	175B	44B	36KB	1	36KB	36KB
users	1	110B	110B	36KB	1	36KB	36KB

Cuenta con 5 colecciones las cuales son artículos, categorías, comentarios, tags y usuarios, cada una dividida de la siguiente manera, con los ejemplos:

Artículos:

```
_id: ObjectId('6910f4cf77e6b878327dca9c')
title : "AudioFlow Lanza los "EchoBuds 3": Cancelación de Ruido que Aprende de …"
date : 2025-11-09T14:08:47.414+00:00
text : "La compañía de audio AudioFlow ha lanzado hoy sus nuevos auriculares i…"
user_id : ObjectId('690e67d3fc47033e857558a8')
▸ categories : Array (2)
▸ tags : Array (4)
```

Categorías:

```
_id: ObjectId('6910f44677e6b878327dca96')
name : "Lanzamientos"
```

```
_id: ObjectId('6910f44e77e6b878327dca97')
name : "IA"
```

Comentarios:

```
_id: ObjectId('6912bf146431e3c04f1c616e')
text : "¡Qué buen artículo! Me sirvió mucho la explicación."
article_id : "6910f4cf77e6b878327dca9c"
user_id : "690e67d3fc47033e857558a860c72b2f9b1d8c001f8e4c6a"
date : "2025-11-10T22:42:00.123Z"
```

Tags:

```
_id: ObjectId('6910f47077e6b878327dca98')
name : "Audifonos"
```

```
_id: ObjectId('6910f47777e6b878327dca99')
name : "2025"
```

```
_id: ObjectId('6910f48177e6b878327dca9a')
name : "Cancelacion de Ruido"
```

Usuarios:

```
_id: ObjectId('690e67d3fc47033e857558a8')
name : "Wikiblogs"
email : "Admin@wikiblogs.com"
password : "Admin1"
role : "admin"
```

Relaciones entre la base de datos.

Se cumplen las relaciones de dos maneras
Guardando el _id de un documento dentro de otro.
Usando el comando $lookup para juntar las colecciones al momento, simulando a un join en sql.

Uno a Muchos

Usuarios ➔ Artículos (Uno-a-Muchos)
Un Usuario puede escribir muchos Artículos.
Pero un Artículo solo tiene un user_id.

Usuarios ➔ Comentarios (Uno-a-Muchos)
Un Usuario tiene muchos Comentarios.
Pero un Comentario tiene un solo user_id.

Artículos ➔ Comentarios (Uno-a-Muchos)
Un Artículo puede tener muchos Comentarios
Pero un Comentario solo pertenece a un artículo (tiene un solo article_id).

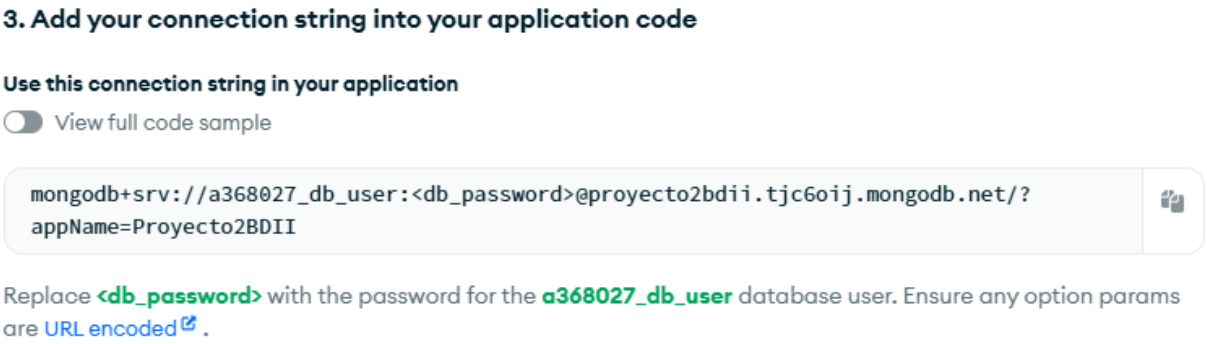
Muchos a Muchos

Artículos ⇔ Categorías (Muchos-a-Muchos)
Un Artículo puede estar en varias Categorías al mismo tiempo.
Y al mismo tiempo, una Categoría puede usar en artículos diferentes.

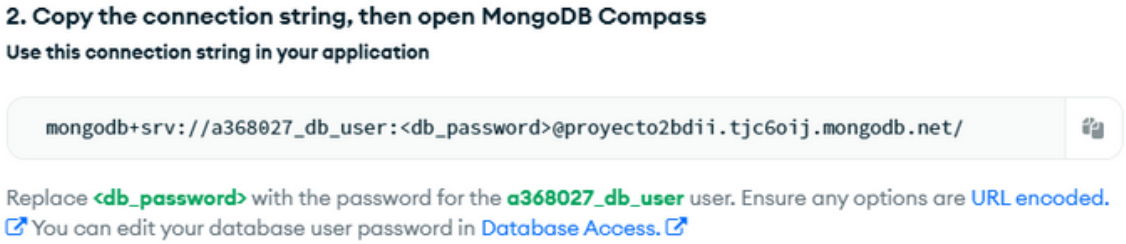
Artículos ⇔ Tags (Muchos-a-Muchos)
Un Artículo puede tener un muchos Tags
Y un Tag se le puede agregar a varios articulos.

Conexión a la base de datos.

La conexión desde MongoDB Atlas se basa en la cadena de conexión del Cluster que nos ofrece Atlas gratuito, el string de conexión creado es la siguiente:
La contraseña se puso cuando se crea la base de datos y el cluster.



También se puede manejar la base de datos desde Compass con una URL que se puede obtener en Atlas, esto para fines de comodidad.



La conexión desde python es de la siguiente manera:

Selecciona la base de datos (Proyecto2BDII) y se le asigna a cada colección a una variable (users_col, articles_col, etc.) para facilitar su uso en las demás funciones.

Incluye un try/except para detener la aplicación si no se puede conectar a la base de datos.

```python
# Conexión
CADENA_ATLAS = "mongodb+srv://a368027_db_user:GGGGP66SFFThwerU@proyecto2bdii.tjc6oij.mongodb.net/Proyecto2BDII?retryWrites=true&w=majority"

try:
    client = MongoClient(CADENA_ATLAS)  #contraseña mongoAtlas: GGGGP66SFFThwerU
    db = client["Proyecto2BDII"]

    users_col = db["users"]
    categories_col = db["categories"]
    tags_col = db["tags"]
    articles_col = db["articles"]
    comments_col = db["comments"]

    client.server_info()
except pymongo.errors.ConnectionFailure as e:
    print(f"Error: No se pudo conectar a MongoDB")
    print(e)
    exit()
```

Funciones CRUD (OperacionesCRUD.py)

Se importa la libreria MongoPy

FUNCIONES DE CREACIÓN (CREATE)

1. **registrar_usuario(nombre, email, password)**
   - Funcionalidad: Inserta un nuevo usuario en la variable users_col.
   - Validación: Antes de insertar, comprueba si el email o el name ya existen en la base de datos.
   - Retorno: Devuelve el inserted_id del nuevo usuario si tiene éxito, o un mensaje de error en texto (string) si el email o nombre ya están en uso.

```python
def registrar_usuario(nombre, email, password):
    if users_col.find_one({"email": email}):
        return "El email registrado."

    if users_col.find_one({"name": nombre}):
        return "Nombre de usuario ya está en uso."

    nuevo_usuario = {
        "name": nombre,
        "email": email,
        "password": password,
        "role": "user"
    }
    result = users_col.insert_one(nuevo_usuario)

    if result.inserted_id:
        return result.inserted_id
    else:
        return "No se pudo registrar al usuario."
```

2. agregar_comentario(article_id_str, user_id_str, texto_comentario)

- Funcionalidad: Inserta un nuevo comentario en la variable comments_col.
- Pasos:
  1. Convierte los IDs de string ("60d...a8b") a ObjectId de MongoDB.
  2. Verifica que el artículo (article_id_str) al que se está comentando exista.
  3. Crea el documento del comentario, incluyendo el texto, los IDs del artículo y usuario, y la fecha actual (datetime.now()).
- Retorno: Devuelve el inserted_id del nuevo comentario.

```python
def agregar_comentario(article_id_str, user_id_str, texto_comentario):
    try:
        obj_article_id = ObjectId(article_id_str)
        obj_user_id = ObjectId(user_id_str)
    except Exception as e:
        print(f" {e}")
        return " "

    # Si el articulo existe
    articulo = articles_col.find_one({"_id": obj_article_id})
    if not articulo:
        return "No se encontró el artículo."

    nuevo_comentario = {
        "text": texto_comentario,
        "article_id": obj_article_id,
        "user_id": obj_user_id,
        "date": datetime.now()
    }
    result = comments_col.insert_one(nuevo_comentario)
    return result.inserted_id
```

3. crear_categoria(nombre)

- Funcionalidad: Inserta una nueva categoría en la categories_col.
- Validación: Comprueba que no exista otra categoría con el mismo nombre.
- Retorno: Devuelve el inserted_id si tiene éxito, o un mensaje de error en texto.

```python
def crear_categoria(nombre):
    # si ya existe
    if categories_col.find_one({"name": nombre}):
        return "Esa categoría ya existe"

    nueva_cat = {"name": nombre}
    result = categories_col.insert_one(nueva_cat)

    if result.inserted_id:
        return result.inserted_id
    else:
        return "No se pudo crear la categoría."
```

4. crear_articulo(user_id, titulo, texto, ids_categorias, ids_tags)

- Funcionalidad: Inserta un nuevo artículo en la articles_col.
- Pasos:
  1. Convierte el user_id y las listas de ids_categorias e ids_tags de string a ObjectId.
  2. Crea el documento del artículo con todos los datos, incluyendo la fecha actual.
- Retorno: Devuelve el inserted_id del nuevo artículo.

```python
def crear_articulo(user_id, titulo, texto, ids_categorias, ids_tags):
    #ObjectId
    try:
        obj_user_id = ObjectId(user_id)
        obj_cats = [ObjectId(cat_id) for cat_id in ids_categorias]
        obj_tags = [ObjectId(tag_id) for tag_id in ids_tags]
    except Exception as e:
        print(f" {e}")
        return " "

    nuevo_articulo = {
        "title": titulo,
        "date": datetime.now(),
        "text": texto,
        "user_id": obj_user_id,
        "categories": obj_cats,
        "tags": obj_tags
    }

    result = articles_col.insert_one(nuevo_articulo)
    return result.inserted_id
```

5. crear_tag(nombre)

- Funcionalidad: Inserta una nueva categoría en la categories_col.
- Validación: Comprueba que no exista otra categoría con el mismo nombre.
- Retorno: Devuelve el inserted_id si tiene éxito, o un mensaje de error en texto.

```python
def crear_tag(nombre):
    #si ya existe
    if tags_col.find_one({"name": nombre}):
        return "Error: Ese tag ya existe, carnal."

    nuevo_tag = {"name": nombre}
    result = tags_col.insert_one(nuevo_tag)

    if result.inserted_id:
        return result.inserted_id
    else:
        return "Error: No se pudo crear el tag."
```

FUNCIONES DE LECTURA (READ)

1. **iniciar_sesion(nombre, email, password)**

- Funcionalidad: Verifica las credenciales de un usuario.
- Lógica: Busca en users_col un documento que coincida exactamente con los tres campos: name, email y password.
- Retorno: Devuelve el documento completo del usuario si lo encuentra, o None si no hay coincidencia.

```python
def iniciar_sesion(nombre, email, password):
    # Busca usuario con los tres campos en la variable users_cpl
    usuario = users_col.find_one({
        "name": nombre,
        "email": email,
        "password": password
    })

    return usuario
```

2.Articulos_blog()

- Funcionalidad: Obtiene todos los artículos mostrarlos en el blog, uniendo datos de 4 colecciones.
- Lógica: Usa un Pipeline de Agregación (como un GROUP BY) de MongoDB:
- $lookup: "Junta" la colección articles con users (para obtener el nombre del autor).
- $lookup: Junta con categories (para obtener los nombres de las categorías).
- $lookup: Junta con tags (para obtener los nombres de los tags).
- $unwind: "Descomprime" la información del autor (ya que $lookup la devuelve en una lista).
- $project: Limpia y renombra los campos (ej. title a titulo) para que sean más fáciles de usar en el HTML.
- $sort: Ordena los artículos por fecha, del más nuevo al más viejo.
- Retorno: Una lista de diccionarios, donde cada diccionario es un artículo con toda la información ya unida.

```python
def obtener_todos_comentarios():
    """
    Trae todos los comentarios con la info del autor y del artículo.
    (La 'R' de CRUD)
    """
    pipeline = [
        # 1. Busca la info del autor
        {"$lookup": {"from": "users", "localField": "user_id", "foreignField": "_id", "as": "autor_info"}},
        # 2. Busca la info del artículo
        {"$lookup": {"from": "articles", "localField": "article_id", "foreignField": "_id", "as": "articulo_info"}},
        # 3. Descomprime los arrays (pa' que sean objetos)
        {"$unwind": {"path": "$autor_info", "preserveNullAndEmptyArrays": True}},
        {"$unwind": {"path": "$articulo_info", "preserveNullAndEmptyArrays": True}},
        {
            # 4. Proyecta solo lo que necesitamos
            "$project": {
                "_id": 1,
                "texto_comentario": "$text",
                "fecha": "$date",
                "autor_nombre": "$autor_info.name",
                "articulo_titulo": "$articulo_info.title"
            }
        },
        # 5. Ordena por fecha (los más nuevos primero)
        {"$sort": {"fecha": -1}}
    ]

    comentarios = list(comments_col.aggregate(pipeline))
    return comentarios
```

FUNCIONES DE LECTURA (READ)

3. obtener_todos_comentarios()

- Funcionalidad: Obtiene todos los comentarios, uniendo datos de 3 colecciones.
- Lógica: Pipeline de Agregación:

1. $lookup: Junta con users para obtener el nombre del autor del comentario.
2. $lookup: Junta con articles para obtener el título del artículo donde se comentó.
3. $unwind: Descomprime los resultados de autor y artículo.
4. $project: Selecciona y renombra los campos necesarios.
5. $sort: Ordena los comentarios por fecha, del más nuevo al más viejo.

- Retorno: Una lista de comentarios con la información del autor y del artículo.

```python
def obtener_todos_comentarios():
    pipeline = [
        # 1. Busca la info del autor
        {"$lookup": {"from": "users", "localField": "user_id", "foreignField": "_id", "as": "autor_info"}},
        # 2. Busca la info del artículo
        {"$lookup": {"from": "articles", "localField": "article_id", "foreignField": "_id", "as": "articulo_info"}},
        # 3. Descomprime los arrays (pa' que sean objetos)
        {"$unwind": {"path": "$autor_info", "preserveNullAndEmptyArrays": True}},
        {"$unwind": {"path": "$articulo_info", "preserveNullAndEmptyArrays": True}},
        {
            # proyecta / Limpia y renombra los campos
            "$project": {
                "_id": 1,
                "texto_comentario": "$text",
                "fecha": "$date",
                "autor_nombre": "$autor_info.name",
                "articulo_titulo": "$articulo_info.title"
            }
        },
        # 5. Ordena por fecha (los más nuevos primero)
        {"$sort": {"fecha": -1}}
    ]

    comentarios = list(comments_col.aggregate(pipeline))
    return comentarios
```

4. obtener_todas_categorias() / obtener_todos_tags()

- Funcionalidad: Obtienen la lista completa de categorías o tags.
- Lógica: Simplemente ejecutan find() sobre la colección correspondiente (categories_col o tags_col).
- Retorno: Una lista de todos los documentos de categorías/tags.

```python
def obtener_todas_categorias():
    return list(categories_col.find())

def obtener_todos_tags():
    return list(tags_col.find())
```

5. obtener_todas_categorias() / obtener_todos_tags()

- Funcionalidad: Obtiene la lista completa de usuarios.
- Lógica: Ejecuta find() sobre users_col y los ordena alfabéticamente por nombre.
- Retorno: Una lista de todos los usuarios.

```python
def obtener_todos_usuarios():
    # .sort("name", 1) para ordenarlos alfabéticamente por nombre
    usuarios = list(users_col.find().sort("name", 1))
    return usuarios
```

6. obtener_articulo_por_id(article_id_str)

- Funcionalidad: Obtiene un único artículo, preparado para un formulario de edición.
- Lógica:
1. Busca el artículo por su _id.
2. Si lo encuentra, convierte sus listas de categories y tags (que son ObjectId) a listas de strings. Para que el formulario HTML pueda marcar los checkboxes correctos.
- Retorno: Un solo documento de artículo con los IDs como strings.

```python
def obtener_articulo_por_id(article_id_str):
    """
    Busca un solo artículo por su ID.
    se usa indirectamente para la opcion de  "Editar".
    """
    try:
        obj_article_id = ObjectId(article_id_str)
    except Exception as e:
        print(f"Error al convertir ID de artículo: {e}")
        return None

    # Busca el artículo.
    articulo = articles_col.find_one({"_id": obj_article_id})

    # convuerte los IDs de categorías y tags a strings
    if articulo:
        articulo['categories'] = [str(cat_id) for cat_id in articulo.get('categories', [])]
        articulo['tags'] = [str(tag_id) for tag_id in articulo.get('tags', [])]

    return articulo
```

7. obtener_categoria_por_id(cat_id_str) / obtener_tag_por_id(tag_id_str)

- Funcionalidad: Obtienen un único tag o categoría por su _id.
- Lógica: Simplemente ejecutan find_one() usando el _id.
- Retorno: El documento del tag/categoría, o None si no existe.

```python
def obtener_categoria_por_id(cat_id_str):
    #busca una categoría por su ID para editarla
    try:
        obj_cat_id = ObjectId(cat_id_str)
    except Exception as e:
        print(f"Error al convertir ID de categoría: {e}")
        return None

    return categories_col.find_one({"_id": obj_cat_id})

def obtener_tag_por_id(tag_id_str):
    #busca una tag por su ID para editarla
    try:
        obj_tag_id = ObjectId(tag_id_str)
    except Exception as e:
        print(f"Error al convertir ID de tag: {e}")
        return None

    return tags_col.find_one({"_id": obj_tag_id})
```

FUNCIONES DE ACTUALIZACION (UPDATE)

1. editar_articulo(article_id_str, titulo, texto, ids_categorias, ids_tags)

- Funcionalidad: Actualiza un artículo existente.
- Lógica:
  1. Define un filtro (buscar por _id).
  2. Define una actualizacion usando $set. Esto solo modifica los campos especificados (title, text, categories, tags) y deja los demás intactos (como user_id y date).
  3. Ejecuta update_one.
- Retorno: True si el artículo fue encontrado y actualizado (o si los datos eran idénticos y no se necesitó cambio), False si no se encontró el artículo.

```python
def editar_articulo(article_id_str, titulo, texto, ids_categorias, ids_tags):
    # IDs de string a ObjectId
    try:
        obj_article_id = ObjectId(article_id_str)
        obj_cats = [ObjectId(cat_id) for cat_id in ids_categorias]
        obj_tags = [ObjectId(tag_id) for tag_id in ids_tags]
    except Exception as e:
        print(f" {e}")
        return False

    filtro = {"_id": obj_article_id} # a quién vamos a buscar
    actualizacion = { # $set actualiza solo estos campos especificados
        "$set": {
            "title": titulo,
            "text": texto,
            "categories": obj_cats,
            "tags": obj_tags
            # No fecha ni user_id
        }}
    try:
        result = articles_col.update_one(filtro, actualizacion)
        # si se modificó algo
        # result.modified_count significa que encontró y actualizó.
        # result.matched_count y modified_count significa que lo encontró, pero los datos eran idénticos y no hubo necesidad de cambiar.

        if result.modified_count == 1 or result.matched_count == 1:
            return True
        else: return False
    except Exception as e:
        print(f"Error {e}")
        return False
```

2. editar_categoria(cat_id_str, nuevo_nombre)

- Funcionalidad: Actualiza el nombre de una categoría.
- Validación: Comprueba si el nuevo_nombre ya existe en otra categoría (usando el operador $ne - "not equal") para evitar duplicados.
- Lógica: update_one y $set para cambiar el campo name.
- Retorno: True si se actualizó, False si no se encontró o si el nombre ya existía.

```python
def editar_categoria(cat_id_str, nuevo_nombre):
    try:
        obj_cat_id = ObjectId(cat_id_str)
    except Exception as e:
        print(f"Error al convertir ID: {e}")
        return False

    # si nuevo nombre ya existe en otra categoría
    if categories_col.find_one({"name": nuevo_nombre, "_id": {"$ne": obj_cat_id}}):
        print("Error: Ya existe otra categoría con ese nombre.")
        return False

    filtro = {"_id": obj_cat_id}
    actualizacion = {"$set": {"name": nuevo_nombre}}

    try:
        result = categories_col.update_one(filtro, actualizacion)
        # Si encontró el documento o modifivo
        if result.modified_count == 1 or result.matched_count == 1:
            return True
        else:
            return False # No la encontró
    except Exception as e:
        print(f"Error al editar categoría: {e}")
        return False
```

3. editar_tag(tag_id_str, nuevo_nombre)

- Funcionalidad: Actualiza el nombre de un tag.
- Lógica: Idéntica a editar_categoria, pero sobre la tags_col.

```python
def editar_tag(tag_id_str, nuevo_nombre):
    try:
        obj_tag_id = ObjectId(tag_id_str)
    except Exception as e:
        print(f"Error al convertir ID: {e}")
        return False

    if tags_col.find_one({"name": nuevo_nombre, "_id": {"$ne": obj_tag_id}}):
        print("Error: Ya existe otro tag con ese nombre.")
        return False

    filtro = {"_id": obj_tag_id}
    actualizacion = {"$set": {"name": nuevo_nombre}}

    try:
        result = tags_col.update_one(filtro, actualizacion)
        if result.modified_count == 1 or result.matched_count == 1:
            return True
        else:
            return False # No lo encontró
    except Exception as e:
        print(f"Error al editar tag: {e}")
        return False
```

FUNCIONES DE ELIMINACION (DELETE)

1. eliminar_usuario(user_id_str)

- Funcionalidad: Elimina un usuario y todo su contenido asociado (borrado en cascada manual).
- Lógica: Este es un proceso de múltiples pasos en orden:

1. Busca todos los artículos de ese usuario (articles_col.find).
2. Borra todos los comentarios hechos en esos artículos (sin importar quién los hizo).
3. Borra todos los artículos de ese usuario (articles_col.delete_many).
4. Borra todos los comentarios que ese usuario hizo en otros artículos (comments_col.delete_many).
5. Finalmente, borra al usuario (users_col.delete_one).

- Retorno: True si el usuario fue eliminado, False si hubo un error o no se encontró.

```python
#Eliminar / Delete
def eliminar_usuario(user_id_str):
    """
    Elimina un usuario y sus relaciones como :
    2. Sus artículos.
    3. Comentarios HECHOS por él.
    """
    # 1. ID del usuario
    try:
        obj_user_id = ObjectId(user_id_str)
    except Exception as e:
        print(f"Error al convertir ID de usuario: {e}")
        return False

    try:
        #Buscar id
        # guardar los IDs de sus artículos para borrar sus comentarios
        articulos_del_usuario_cursor = articles_col.find({"user_id": obj_user_id}, {"_id": 1})
        lista_ids_articulos = [articulo['_id'] for articulo in articulos_del_usuario_cursor]

        if lista_ids_articulos:
            # Borrar  los comentarios de esos artículos de el mismo
            print(f"Borrando comentarios de {len(lista_ids_articulos)} artículos...")
            comments_col.delete_many({"article_id": {"$in": lista_ids_articulos}})

        # Borrar todos los artículos del usuario
        print(f"Borrando artículos del usuario {user_id_str}...")
        articles_col.delete_many({"user_id": obj_user_id})

        # Borrar todos los comentarios que el usuario escribió (en otros blgs)
        print(f"Borrando comentarios hechos por {user_id_str}...")
        comments_col.delete_many({"user_id": obj_user_id})
        # Borrar al usuario
        print(f"Borrando al usuario {user_id_str}...")
        result = users_col.delete_one({"_id": obj_user_id})

        # 6. Verificamos si se borró
        if result.deleted_count == 1:
            return True
        else:
            return False # No encontró al usuario

    except Exception as e:
        print(f"Error durante el borrado en cascada: {e}")
        return False
```

2. eliminar_articulo(article_id_str)

- Funcionalidad: Elimina un artículo y sus comentarios
- Lógica:
  1. Primero, borra todos los comentarios donde article_id coincida (comments_col.delete_many).
  2. Después, borra el artículo (articles_col.delete_one).
- Retorno: True si el artículo fue eliminado.

```python
def eliminar_articulo(article_id_str):
    #Elimina un artículo y TODOS sus comentarios asociados.

    try:
        obj_article_id = ObjectId(article_id_str)
    except Exception as e:
        print(f"Error al convertir ID de artículo: {e}")
        return False

    # Antes de borrar el artículo, se borran todos los comentatios
    try:
        comments_col.delete_many({"article_id": obj_article_id})
    except Exception as e:
        print(f" {e}")
    #Borrar art
    result = articles_col.delete_one({"_id": obj_article_id})

    # 4. Verificamos si se borró
    if result.deleted_count == 1:
        return True
    else:
        return False
```

3. eliminar_comentario_individual(comment_id_str)

- Funcionalidad: Elimina un solo comentario.
- Lógica: Ejecuta delete_one en comments_col usando el _id del comentario.
- Retorno: True si se eliminó (si deleted_count == 1).

```python
def eliminar_comentario_individual(comment_id_str):
    try:
        obj_comment_id = ObjectId(comment_id_str)
    except Exception as e:
        print(f"Error al convertir ID de comentario: {e}")
        return False

    try:
        result = comments_col.delete_one({"_id": obj_comment_id})
        return result.deleted_count == 1
    except Exception as e:
        print(f"Error al eliminar comentario: {e}")
        return False
```

3. eliminar_categoria(cat_id_str)

- Funcionalidad: Elimina una categoría y la "desliga" de todos los artículos.
- Lógica:
  1. Usa articles_col.update_many con el operador $pull. Esto saca el ObjectId de la categoría del array categories en todos los artículos que la contenían.
  2. Después de limpiar las referencias, borra la categoría de categories_col.
- Retorno: True si la categoría fue eliminada.

```python
def eliminar_categoria(cat_id_str):
    #Elimina una categoría Y la quita de todos los artículos

    try:
        obj_cat_id = ObjectId(cat_id_str)
    except Exception as e:
        print(f"Error al convertir ID: {e}")
        return False

    try:
        # $pull actualiza y le quita elemento al array 'categories'
        print(f"Quitando categoría {cat_id_str} de todos los artículos")
        articles_col.update_many(
            {"categories": obj_cat_id}, # Busca artículos que la tengan
            {"$pull": {"categories": obj_cat_id}}
        )

        #  borrar la categoría
        print(f"Borrando categoría {cat_id_str}...")
        result = categories_col.delete_one({"_id": obj_cat_id})

        return result.deleted_count == 1

    except Exception as e:
        print(f"Error al eliminar categoría: {e}")
        return False
```

4. eliminar_tag(tag_id_str)

- Funcionalidad: Elimina un tag y lo "desliga" de todos los artículos.
- Lógica: Idéntica a eliminar_categoria, pero usa $pull sobre el array tags en articles_col.
- Retorno: True si el tag fue eliminado.

```python
def eliminar_tag(tag_id_str):
    #Elimina un tag Y lo quita de todos los artículos

    try:
        obj_tag_id = ObjectId(tag_id_str)
    except Exception as e:
        print(f"Error al convertir ID: {e}")
        return False

    try:
        #
        print(f"Quitando tag {tag_id_str} de todos los artículos...")
        articles_col.update_many(
            {"tags": obj_tag_id}, # Busca artículos que lo tengan
            {"$pull": {"tags": obj_tag_id}}
        )

        # borrar el tag
        print(f"Borrando tag {tag_id_str}...")
        result = tags_col.delete_one({"_id": obj_tag_id})

        return result.deleted_count == 1

    except Exception as e:
        print(f"Error al eliminar tag: {e}")
        return False
```

Framework web: Flujo de peticiones usando Flask.

El sistema opera bajo un patrón de diseño Modelo-Vista-Controlador.

Cada acción del usuario, ya sea agregar, editar o eliminar, sigue un ciclo de peticiones entre estos tres componentes.

Modelo con la BDD(CRUD):
Es el archivo **operacionesCRUD.py**. Es el único componente responsable de la lógica de con la base de datos MongoDB.

Vista (Interfaz de Usuario):
Son los archivos .html en la carpeta **templates/**. Su única función es mostrar datos al usuario (p.ej., {{ categoria.name }}) y capturar nuevas entradas (p.ej., <form>).

Controlador (Lógica de Aplicación): Son los archivos Python en la carpeta **controladores/** (p.ej., categorias.py, articulos.py). Actúan como el intermediario: reciben peticiones del usuario, solicitan datos al Modelo y deciden qué Vista mostrar.

Flujo de Solicitud de Datos (Método GET)

Este es el proceso para mostrar una página:

1. Petición de Usuario: El usuario hace clic en un enlace ("Editar"). El navegador envía una petición HTTP GET a una URL definida (p.ej., /admin/categoria/editar/<id>).
2. Enrutamiento (Flask): Flask intercepta la URL y la dirige a la función correspondiente en el Controlador (p.ej., admin_editar_categoria_ruta en categorias.py).
3. Consulta al Modelo: El Controlador llama a la función de lectura necesaria del Modelo CRUD(p.ej., obtener_categoria_por_id(id)) para buscar los datos en la base de datos.
4. Respuesta del Modelo: El Modelo devuelve los datos solicitados (el documento de la categoría) al Controlador.
5. Renderizado de Vista: El Controlador toma esos datos y los "renderiza" o inyecta en una plantilla de la Vista (p.ej., admin_editar_categoria.html). El HTML resultante se envía al navegador, y el usuario ve el formulario con los datos cargados.

Flujo de Procesamiento de Datos (Método POST)

Este es el proceso para procesar una acción (guardar un cambio, crear un nuevo elemento o eliminar).

1. Envío de Formulario: El usuario presiona un botón (p.ej., "Confirmar Cambios"). El navegador empaqueta los datos del formulario y envía una petición HTTP POST a la URL definida en el <form>.
2. Enrutamiento (Flask): Flask dirige esta petición POST a la función correspondiente en el Controlador (a menudo la misma función que el GET, pero esta vez detecta request.method == "POST").
3. Extracción de Datos: El Controlador extrae los datos enviados del formulario (p.ej., request.form.get('nombre')).
4. Acción en Modelo: El Controlador llama a la función de escritura del Modelo (p.ej., editar_categoria(id, nuevo_nombre)), pasando los nuevos datos.
5. Respuesta del Modelo: El Modelo ejecuta la operación en MongoDB (p.ej., update_one) y devuelve un estado de éxito (p.ej., True) al Controlador.
6. Redirección: Al recibir el éxito, el Controlador no renderiza una vista. En su lugar, emite una redirección (redirect). Esto le ordena al navegador que debe solicitar una nueva página (generalmente, la lista principal, p.ej., /admin/categorias).
7. Ciclo Completo: Esta redirección inicia un nuevo flujo GET (paso 1 del proceso anterior), lo que garantiza que el usuario vea la lista actualizada con los cambios recién guardados.

Correr servicio y app