PRÁCTICA 5

APLICACIÓN PARA GESTIÓN DE UNA LIBRERÍA

Equipo:
Beltrán Saucedo Axel Alejandro
Cerón Samperio Lizeth Montserrat
Higuera Pineda Angel Abraham
Lorenzo Silva Abad Rey
4BV1.

ESCUELA SUPERIOR DE CÓMPUTO

TECNOLOGÍAS PARA EL DESARROLLO DE APLICACIONES WEB

31/10/2025

Índice

1. Introducción **2**

2. Fundamentos Teóricos **2**

3. Diagrama UML **6**

4. Implementación **7**

Referencias **29**

1. Introducción

Planteamiento del problema

Se busca crear una API web cuyo propósito sea gestionar un inventario y optimizar envíos con ayuda del algoritmo genético simple. Para la gestión del inventario, se debe garantizar lo siguiente:

- Permitir crear, leer, actualizar y borrar items, categorías y envíos en una base de datos.

Para la optimización de envíos, se debe garantizar lo siguiente:

- Ofrecer un endpoint que utilice el AGS para resolver el problema de la mochila. Que calcule la combinación de items que maximiza la ganancia total de un envío sin exceder una capacidad de peso determinada.

Propuesta de solución

La solución propuesta es el desarrollo de una API con FastAPI para la lógica de negocio y SQLModel para la gestión de la base de datos. Componentes clave de la solución:

- **SQLModel:** Para definir la estructura de la base de datos estableciendo categoría, item y envío como entidades principales.
- **endpoints (URLs):** Para crear, leer, actualizar y borrar items, categorías y envíos.
- **Algoritmo Génetico:** Realizará el calculo. Simulando un proceso de evolución para encontrar la mejor combinación de items que da la mayor ganancia total sin superar la capacidad.

2. Fundamentos Teóricos

En el desarrollo de esta práctica, se pondrán en uso conceptos y teconologías ya vistas; refinandolos de manera que sea entendible para el cliente, esto en forma de un inventario de biblioteca.

Entorno de Trabajo y Herramientas Principales

- **Python:** Es el lenguaje de programación sobre el que se construye toda la lógica de la aplicación. Su sencilla sintaxis y su amplia cantidad de librerías lo hacen ideal para el desarrollo web.[2]
- **FastAPI:** Framework web moderno para construir APIs con Python. Sus características más destacadas son la rapidez, la validación de datos automática mediante Pydantic y la generación de documentación interactiva , que fue crucial para probar los endpoints de nuestra API.
- **Uvicorn:** Es un servidor ASGI ultrarrápido, utilizado para ejecutar la aplicación FastAPI. Permite que la API maneje múltiples peticiones de forma asíncrona, mejorando el rendimiento.[3]

Persistencia de Datos

La persistencia de datos es la capacidad de un sistema para poder conservar la información posterior de la duración de una sola ejecución. En nuestra practica pasada, los datos eran volátiles, ya que se almacenaban en una lista en memoria. En esta práctica, se implementa la persistencia a través de los siguientes componentes:

- **SQLite:** Es un motor de base de datos relacional, autocontenido y que no requiere un servidor. Almacena toda la base de datos en un único archivo (en nuestro caso, `database.db`). Es ideal para desarrollo y aplicaciones de pequeña a mediana escala por su simplicidad y portabilidad.
- **SQLAlchemy:** Es una librería que funciona como un Mapeador Objeto-Relacional (ORM). Un ORM es una técnica que actúa como un "traductor.^entre el código orientado a objetos y las tablas de una base de datos relacional. Permite manipular la base de datos utilizando código Python en lugar de escribir consultas SQL directamente.
- **SQLModel:** Es una librería que combina **SQLAlchemy** y **Pydantic**, creada por el mismo autor de FastAPI. Permite definir la estructura de los datos, las validaciones y el esquema de la base de datos en una sola clase, reduciendo la duplicación de código y simplificando el desarrollo. En nuestra práctica, las clases como `Item` son modelos de SQLModel que representan tanto la tabla en la base de datos como los datos que la API recibe y envía.[4]

Relaciones en Bases de Datos

- **Relación Unívoca:** Cada valor de clave primaria se relaciona con sólo un registro en la tabla relacionada.
- **Uno a varios:** La tabla de claves primaria sólo contiene un registro que se relaciona con ninguno, uno o varios registros en la tabla relacionada.
- **Varios a varios:** Cada registro en ambas tablas puede estar relacionado con varios registros en la otra tabla. Este tipo de relaciones requieren una tercera tabla, denominada tabla de enlace o asociación, porque los sistemas relacionales no pueden alojar directamente la relación. [5]

API CRUD

Una API CRUD es una interfaz de programación que permite Crear, Leer, Actualizar y Borrar datos. [6]

Búsqueda por parámetros.

La búsqueda por parámetros consiste en consultar datos específicos en la base de datos utilizando ciertos "filtros.º çriterios". En la API se implementará mediante los endpoints:

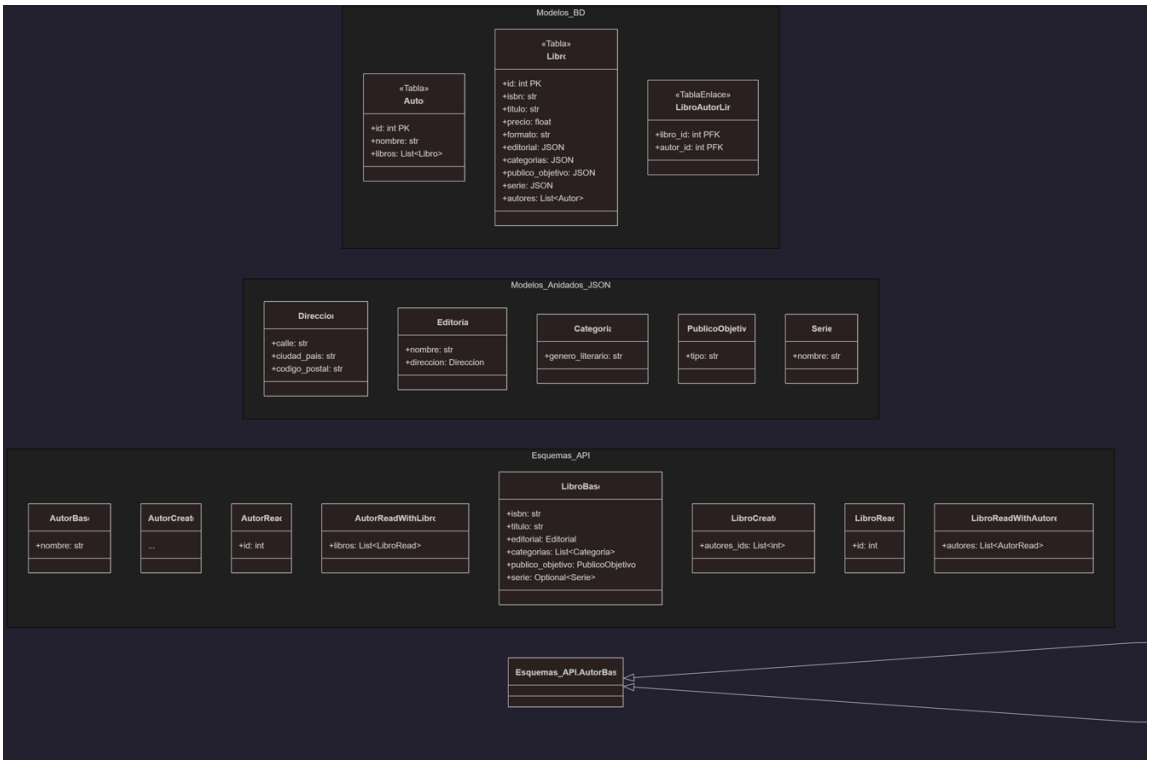
- **Libros del mismo autor.** Que permitirá obtener todos los libros escritos por un mismo autor.
- **Libros por categoría.** Que nos permitirá obtener todos los libros que pertenecen a una categoría específica.
- **Libros por serie.** Que nos permitirá obtener todos los libros pertenecientes a una serie.
- **Libros por público objetivo.** Que nos permitirá obtener los libros según el público al que van dirigidos.

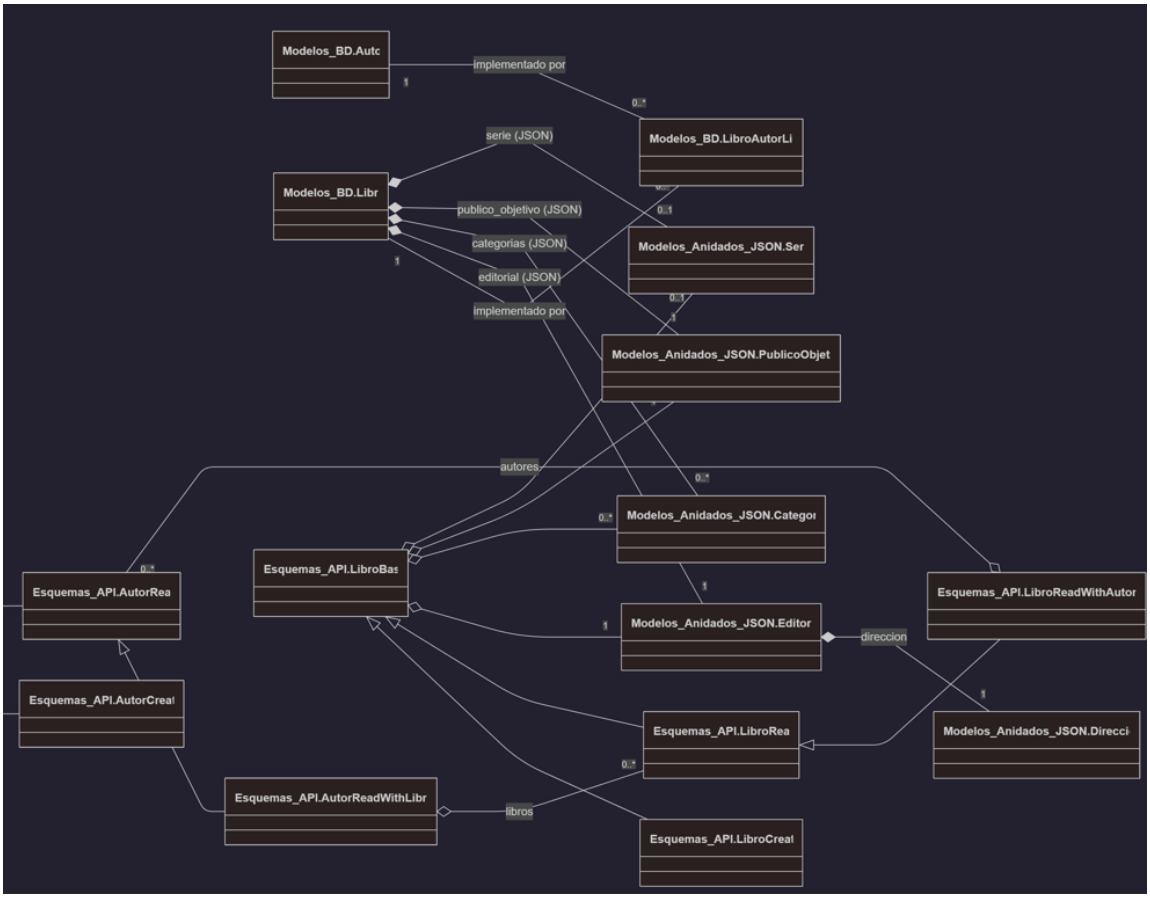
Digitalización de un catálogo.

Se trata de un proceso para la conversión de un catálogo de libros físicos a una búsqueda por consultas en un formato digital. Tomando en cuenta los siguientes aspectos de cada libro:

- Titulo
- ISBN
- Autor(es)
- Editorial
- Año de publicación
- Páginas
- Categorías
- Precio
- Formato (físico o digital)
- Público objetivo (Mayores de edad o menores de edad)
- Serie (si pertenece a una serie de libros)

3. Diagrama UML





4. Implementación

Código:

Para esta practica se nos solicito realizar una aplicacion encargada de gestionar una libreria, para lo cual se organizo el programa en varias carpetas, siguiendo la estructura que se nos dio en clase y con un orden basado en la importancia de cada archivo

Arquitectura:

- **Modelos:** Define las tablas y relaciones en la base de datos utilizando SQLModel.
- **Esquemas:** Define los modelos de datos y las reglas de validación.
- **Servicios:** Contiene la configuración y gestión de la base de datos.
- **Rutas:** Define los endpoints públicos de la API.

Para iniciar con el desarrollo de la aplicación, se creó el archivo principal `main.py`, que configura la aplicación FastAPI y monta los routers de las diferentes entidades (autores, editoriales, público objetivo y series).

Al tener el main para inicar con el programa, debemos continuar con la creacion de la base de datos que se encuentra en la carpeta `Servicios`, ya que sin esto no podriamos almacenar ningun dato.

`Servicios/database.py`: Gestión de la Base de Datos

Este módulo es fundamental para la persistencia de datos.

- **Configuración**: Establece la conexión con la base de datos. Se define la `DATABASE_URL` para utilizar un archivo local de **SQLite** (`libreria.db`).
- **Motor (`engine`)**: Se crea el `engine` (motor) de SQLModel, que sirve como el punto de acceso centralizado y el gestor de la comunicación entre la aplicación y la base de datos.
- **Gestión de Sesiones (`get_session`)**: Expone la función `get_session` como una **dependencia** de FastAPI. Su responsabilidad es ïnyectarüna sesión de base de datos activa en cada *endpoint* que la requiera.

Al tener el çreador"de la base de datos, procedemos a definir la estructura de la misma, para lo cual se creo la carpeta `Modelos`.

`Modelos/modelos.py`: Definición de Tablas y Relaciones

Este archivo es la dependenica cero del programa y la base de datos, es decir no podemos crear/escribir un servicio sin antes tener la estructura de la base de datos definida.

Más que nada este archivo se encarga de pasar la explicacion que se haria en una base de datos de cada tabla y sus relaciones a codigo python, para lo cual se utilizan las clases de `SQLModel`.

Dentro del archivo se encuentra la definicio de las tablas en modo de clases, donde cada clase representa una tabla en la base de datos y sus atributos representan las columnas de dicha tabla:

- **LibroAutorLink**: Tabla de enlace para la relación muchos a muchos entre libros y autores.

```python
# --- Tablas Link (Many-to-Many) ---
#Define la tabla de enlace (asociativa) para la relacion Libro <-> Autor.
class LibroAutorLink(SQLModel, table=True):
    #Define el campo 'libro_id' como clave foranea a 'libro.id' y parte de la clave primaria.
    libro_id: Optional[int] = Field(
        default=None, foreign_key="libro.id", primary_key=True
    )
    #Define el campo 'autor_id' como clave foranea a 'autor.id' y parte de la clave primaria.
    autor_id: Optional[int] = Field(
        default=None, foreign_key="autor.id", primary_key=True
    )
```

- **LibroCategoriaLink**: Tabla de enlace para la relación muchos a muchos entre libros y categorías.

```python
#Define la tabla de enlace (asociativa) para la relacion Libro <-> Categoria.
class LibroCategoriaLink(SQLModel, table=True):
    #Define el campo 'libro_id' como clave foranea a 'libro.id' y parte de la clave primaria.
    libro_id: Optional[int] = Field(
        default=None, foreign_key="libro.id", primary_key=True
    )
    #Define el campo 'categoria_id' como clave foranea a 'categoria.id' y parte de la clave primaria.
    categoria_id: Optional[int] = Field(
        default=None, foreign_key="categoria.id", primary_key=True
    )
```

- **Categoria**: Tabla para almacenar las categorías de los libros.

```python
#Define el modelo de la tabla 'categoria'.
class Categoria(SQLModel, table=True):
    #Define la clave primaria 'id'.
    id: Optional[int] = Field(default=None, primary_key=True)

    # CAMBIO: Estandarizado a 'nombre' y añadido 'unique=True'
    #Define el campo 'nombre', indexado y con restriccion unica.
    nombre: str = Field(index=True, unique=True)
    #Define el campo 'descripcion' como un string opcional.
    descripcion: Optional[str] = None

    #Define la relacion muchos-a-muchos con 'Libro', usando 'LibroCategoriaLink' como tabla de enlace.
    libros: List["Libro"] = Relationship(
        back_populates="categorias", link_model=LibroCategoriaLink
    )
```

- **Direccion**: Tabla para almacenar las direcciones de las editoriales.

```python
# --- Tablas de Entidad ---
#Define el modelo de la tabla 'direccion' en la BD.
class Direccion(SQLModel, table=True):
    #Define la clave primaria 'id' como un entero opcional autogenerado.
    id: Optional[int] = Field(default=None, primary_key=True)
    #Define el campo 'calle' como un string.
    calle: str
    #Define el campo 'ciudad_pais' como un string.
    ciudad_pais: str
    #Define el campo 'codigo_postal' como un string.
    codigo_postal: str

    #Define la relacion inversa (uno-a-uno) con 'Editorial'.
    editorial: Optional["Editorial"] = Relationship(back_populates="direccion")
```

- **Editorial**: Tabla para almacenar las editoriales, con una relación uno a uno con `Direccion`.

```python
#Define el modelo de la tabla 'editorial'.
class Editorial(SQLModel, table=True):
    #Define la clave primaria 'id'.
    id: Optional[int] = Field(default=None, primary_key=True)
    #Define el campo 'nombre' como un string.
    nombre: str

    #Define la clave foranea 'direccion_id' que apunta a 'direccion.id'.
    direccion_id: int = Field(foreign_key="direccion.id")
    #Define la relacion (uno-a-uno) con 'Direccion'.
    direccion: Direccion = Relationship(back_populates="editorial")

    #Define la relacion (uno-a-muchos) con 'Libro' (una editorial tiene muchos libros).
    libros: List["Libro"] = Relationship(back_populates="editorial")
```

- **PublicoObjetivo**: Tabla para almacenar los públicos objetivo de los libros.

```python
#Define el modelo de la tabla 'publicoobjetivo'.
class PublicoObjetivo(SQLModel, table=True):
    #Define la clave primaria 'id'.
    id: Optional[int] = Field(default=None, primary_key=True)
    #Define el campo 'tipo' (ej. +18, Publico en general), y lo marca como indexado.
    tipo: str = Field(index=True) # +18, Publico en general, Dinámico

    #Define la relacion (uno-a-muchos) con 'Libro' (un publico tiene muchos libros).
    libros: List["Libro"] = Relationship(back_populates="publico_objetivo")
```

- **Serie**: Tabla para almacenar las series de libros.

```python
#Define el modelo de la tabla 'serie'.
class Serie(SQLModel, table=True):
    #Define la clave primaria 'id'.
    id: Optional[int] = Field(default=None, primary_key=True)
    #Define el campo 'nombre' como string y lo marca como indexado.
    nombre: str = Field(index=True)

    #Define la relacion (uno-a-muchos) con 'Libro' (una serie tiene muchos libros).
    libros: List["Libro"] = Relationship(back_populates="serie")
```

- **Autor**: Tabla para almacenar los autores de los libros.

```python
#Define el modelo de la tabla 'autor'.
class Autor(SQLModel, table=True):
    #Define la clave primaria 'id'.
    id: Optional[int] = Field(default=None, primary_key=True)
    #Define el campo 'nombre' como string y lo marca como indexado.
    nombre: str = Field(index=True)

    #Define la relacion muchos-a-muchos con 'Libro', usando 'LibroAutorLink' como tabla de enlace.
    libros: List["Libro"] = Relationship(
        back_populates="autores", link_model=LibroAutorLink
    )
```

- **Libro**: Tabla principal para almacenar los libros, con relaciones muchos a muchos con `Autor` y `Categoria`, y relaciones uno a muchos con `Editorial`, `PublicoObjetivo` y `Serie`. En este caso la tabla `Libro` contiene claves foráneas que referencian a las tablas relacionadas. Además en el apartado de ISBN se especifica que es un campo único para evitar duplicados y se tiene una regla que permite crear de forma automática un ISBN si no se proporciona uno al crear un libro. Aunque esto no es una práctica común en el mundo real, se implementó como algo extra.

```python
#Define el modelo de la tabla principal 'libro'.
class Libro(SQLModel, table=True):
    #Define la clave primaria 'id'.
    id: Optional[int] = Field(default=None, primary_key=True)

    #Define el campo 'isbn' como un string.
    isbn: str = Field(
        #Usa 'uuid.uuid4' para generar un ISBN por defecto si no se proporciona uno.
        default_factory=lambda: str(uuid.uuid4()),
        #Asegura que el ISBN sea unico en la tabla.
        unique=True,
        #Crea un indice en este campo para busquedas rapidas.
        index=True
    )
    #Define el campo 'titulo' como un string.
    titulo: str
    #Define el campo 'edicion' como un string opcional.
    edicion: Optional[str] = None
    #Define el campo 'ano_publicacion' como un entero opcional.
    ano_publicacion: Optional[int] = None
    #Define el campo 'paginas' como un entero opcional.
    paginas: Optional[int] = None
    #Define el campo 'precio' como un numero de punto flotante.
    precio: float
    #Define el campo 'formato' (ej. Fisico o digital).
    formato: str

    # --- Relaciones (Foreign Keys) ---
    #Define la clave foranea 'editorial_id' (opcional) que apunta a 'editorial.id'.
    editorial_id: Optional[int] = Field(default=None, foreign_key="editorial.id")
    #Define la relacion (muchos-a-uno) con 'Editorial'.
    editorial: Optional[Editorial] = Relationship(back_populates="libros")

    #Define la clave foranea 'publico_objetivo_id' (opcional).
    publico_objetivo_id: Optional[int] = Field(default=None, foreign_key="publicoobjetivo.id")
    #Define la relacion (muchos-a-uno) con 'PublicoObjetivo'.
    publico_objetivo: Optional[PublicoObjetivo] = Relationship(back_populates="libros")

    #Define la clave foranea 'serie_id' (opcional).
    serie_id: Optional[int] = Field(default=None, foreign_key="serie.id")
    #Define la relacion (muchos-a-uno) con 'Serie'.
    serie: Optional[Serie] = Relationship(back_populates="libros")

    # --- Relaciones (Many-to-Many) ---
    #Define la relacion muchos-a-muchos con 'Autor', usando 'LibroAutorLink'.
    autores: List[Autor] = Relationship(
        back_populates="libros", link_model=LibroAutorLink
    )
    #Define la relacion muchos-a-muchos con 'Categoria', usando 'LibroCategoriaLink'.
    categorias: List[Categoria] = Relationship(
        back_populates="libros", link_model=LibroCategoriaLink
    )
```

Una vez terminado el modelo de la base de datos, procedemos a definir los esquemas de datos que se utilizarán para validar las entradas y salidas de la API, para lo cual se creo la carpeta `Esquemas`.

`Esquemas/esquemas.py`: Definición de Modelos de Datos

Este archivo define la estructura y las reglas de validación para todos los datos que maneja la API, utilizando las clases de `SQLModel`.

- **`...Base` (Ej. `CategoriaBase`)**: Define los atributos comunes y los tipos de datos de una entidad.
- **`...Crear` (Ej. `AutorCreacion`)**: Esquemas utilizados para validar los datos JSON que **ingresan** a la API durante las operaciones de creación (`POST`).
- **`...Leer` (Ej. `AutorLeer`)**: Esquemas que definen el formato de los datos que la API **devuelve** como respuesta. Comúnmente añaden el `id` generado por la BD.
- **`...Actualizar` (Ej. `EditorialActualizar`)**: Esquemas específicos para operaciones `PATCH`, donde todos sus campos son `Optional` para permitir actualizaciones parciales.

```python
# --- Esquemas para Categoria ---
#Define el modelo base para 'Categoria' con los campos comunes.
class CategoriaBase(SQLModel):
    #Define el campo 'nombre' como un string (requerido).
    nombre: str
    #Define el campo 'descripcion' como un string opcional, por defecto Nulo.
    descripcion: Optional[str] = None

#Define el esquema para CREAR una Categoria (lo que la API recibe en un POST).
class CategoriaCrear(CategoriaBase):
    #Hereda todos los campos de CategoriaBase y no añade nuevos.
    pass

#Define el esquema para LEER una Categoria (lo que la API devuelve en un GET).
class CategoriaLeer(CategoriaBase):
    #Añade el 'id' a la respuesta, ya que este es generado por la BD.
    id: int

# --- Esquemas para Autor ---
#Define el modelo base para 'Autor'.
class AutorBase(SQLModel):
    #Define el campo 'nombre' como un string.
    nombre: str

#Define el esquema para CREAR un Autor.
class AutorCreacion(AutorBase):
    #No necesita campos adicionales.
    pass

#Define el esquema para LEER un Autor.
class AutorLeer(AutorBase):
    #Añade el 'id' generado por la BD a la respuesta.
    id: int

# --- Esquemas para Direccion ---
#Define el modelo base para 'Direccion'.
class DireccionBase(SQLModel):
    #Define el campo 'calle' como un string.
    calle: str
    #Define el campo 'ciudad_pais' como un string.
    ciudad_pais: str
    #Define el campo 'codigo_postal' como un string.
    codigo_postal: str

#Define el esquema para CREAR una Direccion.
class DireccionCrear(DireccionBase):
    #No necesita campos adicionales.
    pass

#Define el esquema para LEER una Direccion.
class DireccionLeer(DireccionBase):
    #Añade el 'id' generado por la BD a la respuesta.
    id: int
```

```python
# --- Esquemas para Editorial ---
#Define el modelo base para 'Editorial'.
class EditorialBase(SQLModel):
    #Define el campo 'nombre' como un string.
    nombre: str

#Define el esquema para CREAR una Editorial.
class EditorialCrear(EditorialBase):
    # Al crear una editorial, creamos su dirección al mismo tiempo
    #Espera recibir un objeto 'Direccion' anidado que cumpla con el esquema 'DireccionCrear'.
    direccion: DireccionCrear

#Define el esquema para LEER una Editorial.
class EditorialLeer(EditorialBase):
    #Añade el 'id' generado por la BD.
    id: int
    #Muestra el objeto 'Direccion' completo (anidado) en la respuesta.
    direccion: DireccionLeer # Anidado

#Define el esquema para ACTUALIZAR una Direccion (PATCH).
class DireccionActualizar(SQLModel):
    """Esquema para actualizar solo algunos campos de una dirección"""
    #Define el campo 'calle' como opcional.
    calle: Optional[str] = None
    #Define el campo 'ciudad_pais' como opcional.
    ciudad_pais: Optional[str] = None
    #Define el campo 'codigo_postal' como opcional.
    codigo_postal: Optional[str] = None

#Define el esquema para ACTUALIZAR una Editorial (PATCH).
class EditorialActualizar(SQLModel):
    """Esquema para actualizar una editorial (PATCH)"""
    #Define el campo 'nombre' como opcional.
    nombre: Optional[str] = None
    #Permite actualizar la direccion (o partes de ella) de forma anidada.
    direccion: Optional[DireccionActualizar] = None

# --- Esquemas para PublicoObjetivo ---
#Define el modelo base para 'PublicoObjetivo'.
class PublicoObjetivoBase(SQLModel):
    #Define el campo 'tipo' (ej. "+18", "Infantil").
    tipo: str

#Define el esquema para CREAR un PublicoObjetivo.
class PublicoObjetivoCrear(PublicoObjetivoBase):
    #No necesita campos adicionales.
    pass

#Define el esquema para LEER un PublicoObjetivo.
class PublicoObjetivoLeer(PublicoObjetivoBase):
    #Añade el 'id' generado por la BD.
    id: int

# --- Esquemas para Serie ---
#Define el modelo base para 'Serie'.
class SerieBase(SQLModel):
    #Define el campo 'nombre' como string.
    nombre: str

#Define el esquema para CREAR una Serie.
class SerieCrear(SerieBase):
    #No necesita campos adicionales.
    pass
```

```python
# --- Esquemas para Libro (ACTUALIZADO) ---
#Define el modelo base para 'Libro'.
class LibroBase(SQLModel):
    #Define el campo 'isbn' como opcional (aunque el modelo de BD lo genera).
    isbn: Optional[str] = None
    #Define el campo 'titulo' como string (requerido).
    titulo: str
    #Define el campo 'edicion' como string opcional.
    edicion: Optional[str] = None
    #Define el campo 'ano_publicacion' como entero opcional.
    ano_publicacion: Optional[int] = None
    #Define el campo 'paginas' como entero opcional.
    paginas: Optional[int] = None
    #Define el campo 'precio' como float (requerido).
    precio: float
    #Define el campo 'formato' (ej. "Físico", "Digital").
    formato: str

#Define el esquema para CREAR un Libro (entrada POST).
class LibroCreacion(LibroBase):
    # === CAMBIO IMPORTANTE ===
    # Ahora recibimos nombres (strings) en lugar de IDs
    #Espera el *nombre* de la editorial (string) en lugar de un ID.
    editorial_nombre: Optional[str] = None
    #Espera el *tipo* del publico (string) en lugar de un ID.
    publico_objetivo_tipo: Optional[str] = None # 'tipo' es el "nombre" de esta entidad
    #Espera el *nombre* de la serie (string) en lugar de un ID.
    serie_nombre: Optional[str] = None

    # Mantenemos el patrón de nombres que ya teníamos
    #Espera una lista de *nombres* de autores (strings).
    autores_nombres: List[str] = []
    #Espera una lista de *nombres* de categorias (strings).
    categorias_nombres: List[str] = []

#Define el esquema base para LEER un Libro (salida GET).
class LibroLeer(LibroBase):
    #Añade el 'id' del libro a la respuesta.
    id: int

#Define un esquema de LECTURA COMPLETA para un Libro.
class LibroLeerCompleto(LibroLeer):
    #Muestra el objeto 'Editorial' completo (anidado).
    editorial: Optional[EditorialLeer] = None
    #Muestra el objeto 'PublicoObjetivo' completo (anidado).
    publico_objetivo: Optional[PublicoObjetivoLeer] = None
    #Muestra el objeto 'Serie' completo (anidado).
    serie: Optional[SerieLeer] = None
    #Muestra una lista de objetos 'Autor' completos (anidado).
    autores: List[AutorLeer] = []
    #Muestra una lista de objetos 'Categoria' completos (anidado).
    categorias: List[CategoriaLeer] = []
```

Para que el programa pueda interactuar con la base de datos, es necesario definir funciones que realicen las operaciones CRUD (Crear, Leer, Actualizar, Borrar) para cada entidad. Estas funciones se encuentran en la carpeta `Servicios`.

Es el apartado más extenso, ya que cada entidad tiene sus propias funciones CRUD, además de funciones adicionales para búsquedas específicas en el caso de los libros.

`Servicios/servicios,py`: Funciones CRUD

Cada entidad tiene una función de busqueda por nombre, ID o mostrar todos. El unico que llega a tener más funciones es el de libros, ya que tiene funciones para buscar por autor, categoria, serie, publico objetivo y todos los libros.
Las funciones más importantes son las de la entidad libro, ya que es el apartado principal del programa.

```python
#Define una funcion de ayuda para buscar un Autor por su nombre.
def get_autor_por_nombre(session: Session, nombre: str) -> Optional[modelo.Autor]:
    #Crea una consulta para seleccionar un Autor donde el nombre coincida.
    statement = select(modelo.Autor).where(modelo.Autor.nombre == nombre)
    #Ejecuta la consulta en la sesion y devuelve el primer resultado (o None).
    return session.exec(statement).first()

#Define una funcion de ayuda para buscar una Categoria por su nombre.
def get_categoria_por_nombre(session: Session, nombre: str) -> Optional[modelo.Categoria]:
    #Crea una consulta para seleccionar una Categoria donde el nombre coincida.
    statement = select(modelo.Categoria).where(modelo.Categoria.nombre == nombre)
    #Ejecuta la consulta y devuelve el primer resultado.
    return session.exec(statement).first()

#Define una funcion de ayuda para buscar una Editorial por su nombre.
def get_editorial_por_nombre(session: Session, nombre: str) -> Optional[modelo.Editorial]:
    #Crea una consulta para seleccionar una Editorial donde el nombre coincida.
    statement = select(modelo.Editorial).where(modelo.Editorial.nombre == nombre)
    #Ejecuta la consulta y devuelve el primer resultado.
    return session.exec(statement).first()

#Define una funcion de ayuda para buscar un PublicoObjetivo por su tipo.
def get_publico_objetivo_por_tipo(session: Session, tipo: str) -> Optional[modelo.PublicoObjetivo]:
    #Crea una consulta para seleccionar un PublicoObjetivo donde el tipo coincida.
    statement = select(modelo.PublicoObjetivo).where(modelo.PublicoObjetivo.tipo == tipo)
    #Ejecuta la consulta y devuelve el primer resultado.
    return session.exec(statement).first()

#Define una funcion de ayuda para buscar una Serie por su nombre.
def get_serie_por_nombre(session: Session, nombre: str) -> Optional[modelo.Serie]:
    #Crea una consulta para seleccionar una Serie donde el nombre coincida.
    statement = select(modelo.Serie).where(modelo.Serie.nombre == nombre)
    #Ejecuta la consulta y devuelve el primer resultado.
    return session.exec(statement).first()
```

```python
#Define el servicio para obtener una lista paginada de todos los Libros.
def get_libros_todos(session: Session, skip: int = 0, limit: int = 10) -> List[modelo.Libro]:
    #Crea una consulta seleccionando Libro, aplicando 'offset' y 'limit'.
    statement = select(modelo.Libro).offset(skip).limit(limit)
    #Ejecuta la consulta y devuelve todos los resultados.
    return session.exec(statement).all()

#Define el servicio para obtener libros filtrados por autor (con JOIN).
def get_libros_por_autor(session: Session, nombre_autor: str, skip: int = 0, limit: int = 10) -> List[modelo.Libro]:
    #Crea una consulta compleja con JOINs.
    statement = (
        #Selecciona 'Libro'
        select(modelo.Libro)
        #Une con la tabla de enlace 'LibroAutorLink'.
        .join(modelo.LibroAutorLink)
        #Une con la tabla 'Autor'.
        .join(modelo.Autor)
        #Filtra donde el nombre del Autor coincida.
        .where(modelo.Autor.nombre == nombre_autor)
        #Aplica paginacion.
        .offset(skip).limit(limit)
    )
    #Ejecuta la consulta y devuelve todos los resultados.
    return session.exec(statement).all()

#Define el servicio para obtener libros filtrados por categoria (con JOIN).
def get_libros_por_categoria(session: Session, genero: str, skip: int = 0, limit: int = 10) -> List[modelo.Libro]:
    #Crea una consulta compleja con JOINs.
    statement = (
        #Selecciona 'Libro'
        select(modelo.Libro)
        #Une con la tabla de enlace 'LibroCategoriaLink'.
        .join(modelo.LibroCategoriaLink)
        #Une con la tabla 'Categoria'.
        .join(modelo.Categoria)
        #Filtra donde el nombre de la Categoria coincida (variable 'genero').
        .where(modelo.Categoria.nombre == genero)
        #Aplica paginacion.
        .offset(skip).limit(limit)
    )
    #Ejecuta la consulta y devuelve todos los resultados.
    return session.exec(statement).all()
```

```python
#Define el servicio para obtener libros filtrados por serie (con JOIN).
def get_libros_por_serie(session: Session, nombre_serie: str, skip: int = 0, limit: int = 10) -> List[modelo.Libro]:
    #Crea una consulta con JOIN (SQLModel infiere el JOIN simple).
    statement = (
        #Selecciona 'Libro'
        select(modelo.Libro)
        #Une con la tabla 'Serie' (usando la FK 'serie_id').
        .join(modelo.Serie)
        #Filtra donde el nombre de la Serie coincida.
        .where(modelo.Serie.nombre == nombre_serie)
        #Aplica paginacion.
        .offset(skip).limit(limit)
    )
    #Ejecuta la consulta y devuelve todos los resultados.
    return session.exec(statement).all()

#Define el servicio para obtener libros filtrados por publico (con JOIN).
def get_libros_por_publico(session: Session, tipo_publico: str, skip: int = 0, limit: int = 10) -> List[modelo.Libro]:
    #Crea una consulta con JOIN.
    statement = (
        #Selecciona 'Libro'
        select(modelo.Libro)
        #Une con la tabla 'PublicoObjetivo' (usando la FK).
        .join(modelo.PublicoObjetivo)
        #Filtra donde el tipo de PublicoObjetivo coincida.
        .where(modelo.PublicoObjetivo.tipo == tipo_publico)
        #Aplica paginacion.
        .offset(skip).limit(limit)
    )
    #Ejecuta la consulta y devuelve todos los resultados.
    return session.exec(statement).all()

# (Añade esto en Servicios/servicios.py)

#Define el servicio para obtener un Libro por su ISBN.
def get_libro_por_isbn(session: Session, isbn: str) -> Optional[modelo.Libro]:
    """Busca un libro por su ISBN."""
    #Crea una consulta para seleccionar un Libro donde el ISBN coincida.
    statement = select(modelo.Libro).where(modelo.Libro.isbn == isbn)
    #Ejecuta la consulta y devuelve el primer resultado (o None).
    libro = session.exec(statement).first()
    #Devuelve el libro encontrado.
    return libro
```

Como se menciono la entidad libro es la más importante, por lo cual solo se mostara la funcion de creacion de libro:

```python
#Define el servicio para crear un Libro (logica compleja de relaciones).
def create_libro(session: Session, libro_create: esquemas.LibroCreacion) -> modelo.Libro:

    # 1. Separar los datos
    #Convierte el esquema de entrada a un diccionario, excluyendo los campos de relacion (nombres).
    libro_data = libro_create.model_dump(exclude={
        "autores_nombres", "categorias_nombres",
        "editorial_nombre", "publico_objetivo_tipo", "serie_nombre"
    })
    #Crea la instancia del modelo Libro solo con los datos base (titulo, precio, etc.).
    db_libro = modelo.Libro(**libro_data)

    # 2. Asignar Editorial por NOMBRE
    #Si se proporciono un nombre de editorial.
    if libro_create.editorial_nombre:
        #Usa la funcion de ayuda para buscar la editorial por su nombre.
        editorial = get_editorial_por_nombre(session, libro_create.editorial_nombre)
        #Si la editorial no se encuentra.
        if not editorial:
            #Lanza un error 404.
            raise HTTPException(status_code=404, detail=f"Editorial '{libro_create.editorial_nombre}' no encontrada")
        #Asigna el objeto 'editorial' completo a la relacion del libro.
        db_libro.editorial = editorial # Asignamos el objeto, no el ID

    # 3. Asignar Publico Objetivo por TIPO
    #Si se proporciono un tipo de publico.
    if libro_create.publico_objetivo_tipo:
        #Usa la funcion de ayuda para buscar el publico por su tipo.
        publico = get_publico_objetivo_por_tipo(session, libro_create.publico_objetivo_tipo)
        #Si no se encuentra.
        if not publico:
            #Lanza un error 404.
            raise HTTPException(status_code=404, detail=f"Público '{libro_create.publico_objetivo_tipo}' no encontrado")
        #Asigna el objeto 'publico' completo a la relacion del libro.
        db_libro.publico_objetivo = publico

    # 4. Asignar Serie por NOMBRE
    #Si se proporciono un nombre de serie.
    if libro_create.serie_nombre:
        #Usa la funcion de ayuda para buscar la serie por su nombre.
        serie = get_serie_por_nombre(session, libro_create.serie_nombre)
        #Si no se encuentra.
        if not serie:
            #Lanza un error 404.
            raise HTTPException(status_code=404, detail=f"Serie '{libro_create.serie_nombre}' no encontrada")
        #Asigna el objeto 'serie' completo a la relacion del libro.
        db_libro.serie = serie
```

```python
# 5. Asignar Autores por NOMBRE
#Itera sobre la lista de nombres de autores proporcionada.
for autor_nombre in libro_create.autores_nombres:
    #Usa la funcion de ayuda para buscar cada autor por su nombre.
    autor = get_autor_por_nombre(session, autor_nombre)
    #Si el autor no se encuentra.
    if not autor:
        #Lanza un error 404.
        raise HTTPException(status_code=404, detail=f"Autor '{autor_nombre}' no encontrado")
    #Anade el objeto 'autor' encontrado a la lista 'autores' del libro (M2M).
    db_libro.autores.append(autor)

# 6. Asignar Categorías por NOMBRE
#Itera sobre la lista de nombres de categorias proporcionada.
for categoria_nombre in libro_create.categorias_nombres:
    #Usa la funcion de ayuda para buscar cada categoria por su nombre.
    categoria = get_categoria_por_nombre(session, categoria_nombre)
    #Si la categoria no se encuentra.
    if not categoria:
        #Lanza un error 404.
        raise HTTPException(status_code=404, detail=f"Categoría '{categoria_nombre}' no encontrada")
    #Anade el objeto 'categoria' encontrado a la lista 'categorias' del libro (M2M).
    db_libro.categorias.append(categoria)

# 7. Guardar en la BD
#Anade el objeto 'db_libro' (con todas sus relaciones) a la sesion.
session.add(db_libro)
#Guarda los cambios (SQLModel/SQLAlchemy se encarga de las tablas de enlace).
session.commit()
#Refresca el objeto para obtener su ID.
session.refresh(db_libro)
#Devuelve el libro recien creado.
return db_libro
```

Al tener los esquemas de datos y sus funciones procedemos a definir los controladores de los endpoints de la API, para lo cual se creo la carpeta `Rutas`.

`Rutas/*.py`: Controladores de Endpoints de la API

Estos archivos definen los *endpoints* (URLs) públicos de la aplicación utilizando `APIRouter` de FastAPI para organizar las rutas de manera modular.

- **`Rutas/autores.py`**: Establece el prefijo `/Autores` e implementa las operaciones `POST` (crear), `GET` (leer) y `DELETE` (eliminar) para los autores.

```python
# --- Rutas para Autores ---
#Define el endpoint POST en la raiz (/Autores/), especificando el modelo de respuesta.
@router.post("/", response_model=esquemas.AutorLeer)
#Define la funcion para crear un autor.
def crear_autor(
    #Define que el cuerpo (body) de la peticion debe seguir el esquema 'AutorCreacion'.
    autor: esquemas.AutorCreacion,
    #Inyecta la dependencia de la sesion de la base de datos.
    session: Session = Depends(get_session)
):
    #Delega la logica de creacion al modulo de 'servicios' y devuelve el resultado.
    return servicios.create_autor(session=session, autor_create=autor)


#Define el endpoint GET en la raiz (/Autores/), respondiendo con una lista de autores.
@router.get("/", response_model=List[esquemas.AutorLeer])
#Define la funcion para leer todos los autores.
def leer_autores(
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' (para paginacion), con valor minimo 0.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' (para paginacion), con minimo 1 y maximo 100.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio para obtener la lista de autores, pasando la paginacion.
    autores = servicios.get_autores_todos(session, skip=skip, limit=limit)
    #Devuelve la lista de autores encontrada.
    return autores

#Define el endpoint DELETE para un autor especifico, esperando un codigo 204 (No Content) al exito.
@router.delete("/{autor_id}", status_code=204)
#Define la funcion para eliminar un autor.
def eliminar_autor(
    #Recibe el 'autor_id' desde la ruta (path parameter).
    autor_id: int,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session)
):
    #Llama al servicio para obtener el autor por su ID.
    db_autor = servicios.get_autor(session, autor_id=autor_id)
    #Si el autor no se encuentra (devuelve None).
    if not db_autor:
        #Lanza un error HTTP 404 (No Encontrado).
        raise HTTPException(status_code=404, detail="Autor no encontrado")

    #Marca el objeto 'db_autor' para ser eliminado.
    session.delete(db_autor)
    #Confirma (guarda) la eliminacion en la base de datos.
    session.commit()
    #Devuelve la respuesta (automaticamente sera 204 No Content, como se definio).
    return
```

- **`Rutas/editoriales.py`**: Gestiona el CRUD completo para `/Editoriales`, incluyendo la ruta `PATCH` para actualizaciones parciales y la lógica `DELETE` para borrar la editorial y su dirección.

```python
#Define el endpoint POST en la raiz (/Editoriales/), especificando el modelo de respuesta.
@router.post("/", response_model=esquemas.EditorialLeer)
#Define la funcion para crear una editorial.
def crear_editorial(
    #Define que el cuerpo (body) de la peticion debe seguir el esquema 'EditorialCrear'.
    editorial: esquemas.EditorialCrear,
    #Inyecta la dependencia de la sesion de la base de datos.
    session: Session = Depends(get_session)
):
    #Delega la logica de creacion (incluyendo la creacion anidada de la direccion) al modulo de 'servicios'.
    return servicios.create_editorial(session=session, editorial_create=editorial)

#Define el endpoint GET en la raiz (/Editoriales/), respondiendo con una lista de editoriales.
@router.get("/", response_model=List[esquemas.EditorialLeer])
#Define la funcion para leer todas las editoriales.
def leer_editoriales_todas(
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' (para paginacion), con valor minimo 0.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' (para paginacion), con minimo 1 y maximo 100.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio para obtener la lista de editoriales, pasando la paginacion.
    editoriales = servicios.get_editoriales_todas(session, skip=skip, limit=limit)
    #Devuelve la lista de editoriales encontrada.
    return editoriales


# (Añade esto en Rutas/editoriales.py, junto a tus otras rutas)

#Define el endpoint PATCH para una editorial especifica, especificando el modelo de respuesta.
@router.patch("/{editorial_id}", response_model=esquemas.EditorialLeer)
#Define la funcion para actualizar una editorial.
def actualizar_editorial_ruta(
    #Recibe el 'editorial_id' desde la ruta (path parameter).
    editorial_id: int,
    #Recibe los datos de actualizacion (del body) que coinciden con el esquema.
    editorial_update: esquemas.EditorialActualizar,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session)
):
    # 1. Busca la editorial
    #Llama al servicio 'get_editorial' para encontrar la editorial por su ID.
    db_editorial = servicios.get_editorial(session, editorial_id=editorial_id)
    #Si el servicio devuelve None (no se encontro).
    if not db_editorial:
        #Lanza un error HTTP 404 (No Encontrado).
        raise HTTPException(status_code=404, detail="Editorial no encontrada")

    # 2. Llama al servicio para que haga la actualización
    #Delega la logica de actualizacion (incluyendo la direccion anidada) al modulo de 'servicios'.
    return servicios.actualizar_editorial(
        #Pasa la sesion actual al servicio.
        session=session,
        #Pasa la editorial ya encontrada al servicio.
        db_editorial=db_editorial,
        #Pasa los datos de actualizacion al servicio.
        editorial_update=editorial_update
    )

#Define el endpoint DELETE para una editorial especifica, esperando un codigo 204 (No Content).
@router.delete("/{editorial_id}",
    #Define el codigo de estado HTTP para una eliminacion exitosa.
    status_code=204
)
```

- **`Rutas/publico_objetivo.py` y `Rutas/series.py`**: Exponen las rutas `POST` y `GET` para crear y leer, respectivamente, las entidades de público objetivo y series.

```python
#Define el endpoint POST en la raiz (/PublicoObjetivo/), especificando el modelo de respuesta.
@router.post("/", response_model=esquemas.PublicoObjetivoLeer)
#Define la funcion para crear un tipo de publico objetivo.
def crear_publico_objetivo(
    #Define que el cuerpo (body) de la peticion debe seguir el esquema 'PublicoObjetivoCrear'.
    publico: esquemas.PublicoObjetivoCrear,
    #Inyecta la dependencia de la sesion de la base de datos.
    session: Session = Depends(get_session)
):
    #Delega la logica de creacion al modulo de 'servicios'.
    return servicios.create_publico_objetivo(session=session, publico_create=publico)

#Define el endpoint GET en la raiz (/PublicoObjetivo/), respondiendo con una lista.
@router.get("/", response_model=List[esquemas.PublicoObjetivoLeer])
#Define la funcion para leer todos los tipos de publico.
def leer_publicos_objetivo_todos(
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' (para paginacion), con valor minimo 0.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' (para paginacion), con minimo 1 y maximo 100.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio para obtener la lista de publicos, pasando la paginacion.
    publicos = servicios.get_publicos_objetivo_todos(session, skip=skip, limit=limit)
    #Devuelve la lista de publicos encontrada.
    return publicos
```

```python
#Define el endpoint POST en la raiz (/Series/), especificando el modelo de respuesta.
@router.post("/", response_model=esquemas.SerieLeer)
#Define la funcion para crear una serie.
def crear_serie(
    #Define que el cuerpo (body) de la peticion debe seguir el esquema 'SerieCrear'.
    serie: esquemas.SerieCrear,
    #Inyecta la dependencia de la sesion de la base de datos.
    session: Session = Depends(get_session)
):
    #Delega la logica de creacion al modulo de 'servicios'.
    return servicios.create_serie(session=session, serie_create=serie)

#Define el endpoint GET en la raiz (/Series/), respondiendo con una lista de series.
@router.get("/", response_model=List[esquemas.SerieLeer])
#Define la funcion para leer todas las series.
def leer_series_todas(
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' (para paginacion), con valor minimo 0.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' (para paginacion), con minimo 1 y maximo 100.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio para obtener la lista de series, pasando la paginacion.
    series = servicios.get_series_todas(session, skip=skip, limit=limit)
    #Devuelve la lista de series encontrada.
    return series
```

- **`Rutas/categorias.py`**: Implementa el CRUD completo para `/Categorias`, similar a `editoriales.py`.

```python
#Define el endpoint POST en la raiz (/Categorias/), especificando el modelo de respuesta.
@router.post("/", response_model=esquemas.CategoriaLeer)
#Define la funcion para crear una categoria.
def crear_categoria(
    #Define que el cuerpo (body) de la peticion debe seguir el esquema 'CategoriaCrear'.
    categoria: esquemas.CategoriaCrear,
    #Inyecta la dependencia de la sesion de la base de datos.
    session: Session = Depends(get_session)
):
    #Delega la logica de creacion (incluyendo la validacion de nombre unico) al modulo de 'servicios'.
    return servicios.create_categoria(session=session, categoria_create=categoria)

#Define el endpoint GET en la raiz (/Categorias/), respondiendo con una lista de categorias.
@router.get("/", response_model=List[esquemas.CategoriaLeer])
#Define la funcion para leer todas las categorias.
def leer_categorias(
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' (para paginacion), con valor minimo 0.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' (para paginacion), con minimo 1 y maximo 100.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio para obtener la lista de categorias, pasando la paginacion.
    categorias = servicios.get_categorias_todos(session, skip=skip, limit=limit)
    #Devuelve la lista de categorias encontrada.
    return categorias

#Define el endpoint DELETE para una categoria especifica, esperando un codigo 204 (No Content) al exito.
@router.delete("/{categoria_id}", status_code=204)
#Define la funcion para eliminar una categoria.
def eliminar_categoria(
    #Recibe el 'categoria_id' desde la ruta (path parameter).
    categoria_id: int,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session)
):
    #Llama al servicio para obtener la categoria por su ID.
    db_categoria = servicios.get_categoria(session, categoria_id=categoria_id)
    #Si la categoria no se encuentra (devuelve None).
    if not db_categoria:
        #Lanza un error HTTP 404 (No Encontrado).
        raise HTTPException(status_code=404, detail="Categoria no encontrada")

    #Marca el objeto 'db_categoria' para ser eliminado.
    session.delete(db_categoria)
    #Confirma (guarda) la eliminacion en la base de datos.
    session.commit()
    #Devuelve la respuesta (automaticamente sera 204 No Content).
    return
```

- **`Rutas/libros.py`**: El archivo más complejo y principal de todos, maneja el CRUD completo para `/Libros` y define múltiples rutas `GET` para búsquedas específicas (por autor, categoría, serie y público objetivo).

```python
# --- Rutas para Libros ---
#Define el endpoint POST en la raiz (/Libros/), especificando el modelo de respuesta (completo).
@router.post("/", response_model=esquemas.LibroLeerCompleto) # Cambiado
#Define la funcion para crear un libro.
def crear_libro(
    #Define que el cuerpo (body) de la peticion debe seguir el esquema 'LibroCreacion'.
    libro: esquemas.LibroCreacion, # Cambiado
    #Inyecta la dependencia de la sesion de la base de datos.
    session: Session = Depends(get_session)
):
    #Delega la logica de creacion (buscar/crear relaciones por nombre) al modulo de 'servicios'.
    return servicios.create_libro(session=session, libro_create=libro)

# 1. Endpoint general con paginación
#Define el endpoint GET en la raiz (/Libros/), respondiendo con una lista de libros completos.
@router.get("/", response_model=List[esquemas.LibroLeerCompleto]) # Cambiado
#Define la funcion para leer todos los libros.
def leer_libros_todos(
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' (para paginacion), con valor minimo 0.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' (para paginacion), con minimo 1 y maximo 100.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio para obtener la lista de libros, pasando la paginacion.
    libros = servicios.get_libros_todos(session, skip=skip, limit=limit)
    #Devuelve la lista de libros encontrada.
    return libros
```

```python
# 1. Consultar libros x autor
#Define el endpoint GET en /por-autor, respondiendo con una lista de libros completos.
@router.get("/por-autor", response_model=List[esquemas.LibroLeerCompleto]) # Cambiado
#Define la funcion para leer libros filtrados por autor.
def leer_libros_por_autor(
    #Recibe el 'nombre_autor' como parametro de consulta (query parameter).
    nombre_autor: str,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' para paginacion.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' para paginacion.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio que contiene la logica de filtrado por autor.
    libros = servicios.get_libros_por_autor(
        session, nombre_autor=nombre_autor, skip=skip, limit=limit
    )
    #Devuelve la lista de libros filtrada.
    return libros

# 2. Libros x categoria
#Define el endpoint GET en /por-categoria.
@router.get("/por-categoria", response_model=List[esquemas.LibroLeerCompleto]) # Cambiado
#Define la funcion para leer libros filtrados por categoria.
def leer_libros_por_categoria(
    #Recibe el 'genero' (nombre de la categoria) como parametro de consulta.
    genero: str, # Este 'genero' es ahora el 'nombre' de la categoría
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' para paginacion.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' para paginacion.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio que contiene la logica de filtrado por categoria.
    libros = servicios.get_libros_por_categoria(
        session, genero=genero, skip=skip, limit=limit
    )
    #Devuelve la lista de libros filtrada.
    return libros
```

```python
# 3. Libros x serie
#Define el endpoint GET en /por-serie.
@router.get("/por-serie", response_model=List[esquemas.LibroLeerCompleto])
#Define la funcion para leer libros filtrados por serie.
def leer_libros_por_serie(
    #Recibe el 'nombre_serie' como parametro de consulta.
    nombre_serie: str,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' para paginacion.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' para paginacion.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio que contiene la logica de filtrado por serie.
    libros = servicios.get_libros_por_serie(
        session, nombre_serie=nombre_serie, skip=skip, limit=limit
    )
    #Devuelve la lista de libros filtrada.
    return libros

# (Añade esto en Rutas/libros.py)

#Define el endpoint GET para buscar un libro por su ISBN (parametro de ruta).
@router.get("/isbn/{isbn}", response_model=esquemas.LibroLeerCompleto)
#Define la funcion para leer un libro por ISBN.
def leer_libro_por_isbn(
    #Recibe el 'isbn' desde la ruta (path parameter).
    isbn: str,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session)
):
    """Obtiene un libro específico por su ISBN."""
    #Llama al servicio que busca el libro por ISBN.
    db_libro = servicios.get_libro_por_isbn(session, isbn=isbn)
    #Si el servicio devuelve None (no se encontro).
    if not db_libro:
        #Lanza un error HTTP 404 (No Encontrado).
        raise HTTPException(
            status_code=404,
            detail="Libro con ese ISBN no encontrado"
        )
    #Devuelve el libro encontrado.
    return db_libro
```

```python
# 4. Libros x público objetivo
#Define el endpoint GET en /por-publico.
@router.get("/por-publico", response_model=List[esquemas.LibroLeerCompleto])
#Define la funcion para leer libros filtrados por publico.
def leer_libros_por_publico(
    #Recibe el 'tipo_publico' como parametro de consulta.
    tipo_publico: str,
    #Inyecta la dependencia de la sesion.
    session: Session = Depends(get_session),
    #Define el parametro de consulta 'skip' para paginacion.
    skip: int = Query(0, ge=0),
    #Define el parametro de consulta 'limit' para paginacion.
    limit: int = Query(10, ge=1, le=100)
):
    #Llama al servicio que contiene la logica de filtrado por publico.
    libros = servicios.get_libros_por_publico(
        session, tipo_publico=tipo_publico, skip=skip, limit=limit
    )
    #Devuelve la lista de libros filtrada.
    return libros
```

Referencias

[1] IBM. (s.f.). ¿Qué es una API REST? Recuperado el 14 de octubre de 2025, de https://www.ibm.com/mx-es/topics/rest-apis

[2] Python Software Foundation. (s.f.). Acerca de Python™. Resumen Ejecutivo. Recuperado el 14 de octubre de 2025, de https://www.python.org/doc/essays/blurb-es/

[3] Ramírez, S. (s.f.). FastAPI. Recuperado el 14 de octubre de 2025, de https://fastapi.tiangolo.com/es/

[4] Ramírez, S. (s.f.). SQLModel. Recuperado el 14 de octubre de 2025, de https://sqlmodel.tiangolo.com/es/

[5] IBM Control Desk. (s. f.). Relaciones en Bases de Datos. Recuperado el 31 de octubre de 2025, de https://www.ibm.com/docs/es/control-desk/7.6.1?topic=structure-database-relationships

[6] Jain, A. (2024, 8 septiembre). What is CRUD API? DEV Community. Recuperado el 31 de octubre de 2025, de https://dev.to/ankitjaininfo/what-is-crud-api-502i

[7] Tema 2. Algoritmos genéticos. (s. f.). Departamento de Ciencias de la Computación E Inteligencia Artificial. Recuperado el 31 de octubre de 2025, de http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t2geneticos