
UNIVERSIDAD TECNOLÓGICA DE XICOTEPEC DE JUÁREZ

Sistema de Gestión de Farmacia (Back-end)

Alumnos:

(220296) José Daniel Loza Marín.

(220186) Luis Abdiel Rivera Gayosso.

(220336) Griselda Cabrera Franco.

(220864) Esaú Vargas Álvarez.

(220418) Jareni Gómez Juan.

Grado: Octavo Cuatrimestre

Grupo: A

Carrera: Ingeniería en Desarrollo y Gestión de Software.

Materia: Administración de Base de datos.

Asesor: Héctor Valderrabano González.

Xicotepec de Juárez Puebla 09 de abril de 2025.

Índice

Introducción.

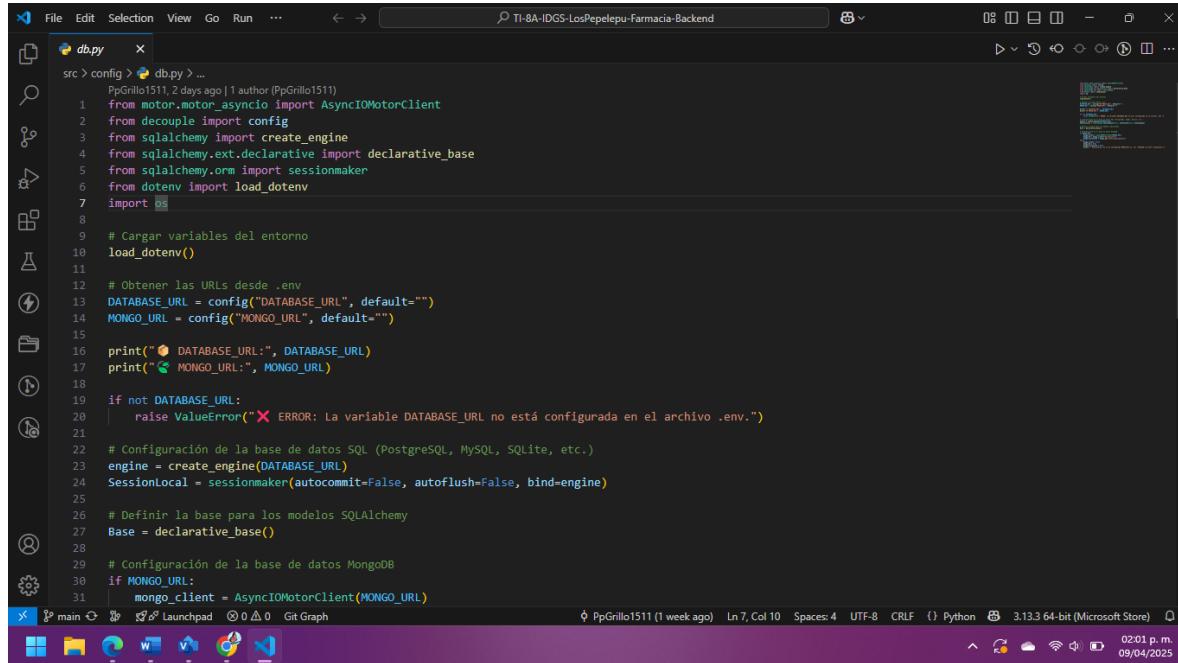
El presente documento describe la estructura, funcionalidades y componentes principales del sistema **back-end** desarrollado para el proyecto **Hospital Las Lomas**, específicamente enfocado en el módulo de **Farmacia**. Este sistema ha sido construido con el objetivo de proporcionar una base robusta, segura y escalable para la gestión eficiente de medicamentos, consumibles, lotes, y procesos de dispensación dentro del entorno hospitalario.

El back-end actúa como el núcleo lógico de la aplicación, permitiendo la comunicación fluida entre la base de datos y el front-end. A través de una API RESTful, se gestionan las operaciones de creación, lectura, actualización y eliminación (CRUD) de los distintos recursos relacionados con la farmacia del hospital. Además, se implementan validaciones, control de acceso, y lógica de negocio para garantizar la integridad y seguridad de los datos médicos y farmacéuticos.

Esta documentación tiene como propósito guiar a los desarrolladores actuales y futuros en la comprensión, mantenimiento y extensión del sistema back-end, proporcionando una visión clara de su arquitectura, rutas, controladores, modelos y dependencias utilizadas.

Estructura.

- **config/db.py:** Configuraciones de la base de datos.



```

File Edit Selection View Go Run ... ← → ⌂ TI-BA-IDGS-LosPepelepu-Farmacia-Backend ⌂ 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
db.py x
src > config > db.py > ...
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
1 from motor.motor_asyncio import AsyncIOMotorClient
2 from decouple import config
3 from sqlalchemy import create_engine
4 from sqlalchemy.ext.declarative import declarative_base
5 from sqlalchemy.orm import sessionmaker
6 from dotenv import load_dotenv
7 import os
8
9 # Cargar variables del entorno
10 load_dotenv()
11
12 # Obtener las URLs desde .env
13 DATABASE_URL = config("DATABASE_URL", default="")
14 MONGO_URL = config("MONGO_URL", default="")
15
16 print("🌐 DATABASE_URL:", DATABASE_URL)
17 print("MongoDB URL:", MONGO_URL)
18
19 if not DATABASE_URL:
20     raise ValueError("🔴 ERROR: La variable DATABASE_URL no está configurada en el archivo .env.")
21
22 # Configuración de la base de datos SQL (PostgreSQL, MySQL, SQLite, etc.)
23 engine = create_engine(DATABASE_URL)
24 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
25
26 # Definir la base para los modelos SQLAlchemy
27 Base = declarative_base()
28
29 # Configuración de la base de datos MongoDB
30 if MONGO_URL:
31     mongo_client = AsyncIOMotorClient(MONGO_URL)

```

Importaciones:

- **motor.motor_asyncio.AsyncIOMotorClient:** Para conectar a MongoDB de forma asíncrona.
- **decouple.config:** Para cargar variables de entorno de forma segura.
- **sqlalchemy:** Para conectar y manejar bases de datos SQL.
- **dotenv:** Para cargar variables de entorno desde un archivo **.env**

Carga de Variables de Entorno:

- Usa **load_dotenv()** para cargar las variables desde un archivo **.env**.
- Obtiene las URLs de la base de datos SQL y MongoDB.

Validación de Variables:

- Verifica que **DATABASE_URL** esté configurada. Si no, lanza un error, lo que ayuda a prevenir problemas durante la conexión.

Configuración de Base de Datos SQL:

- Utiliza **create_engine()** para crear el motor de la base de datos y **sessionmaker()** para generar sesiones que manejarán las transacciones.

Declaración de Modelos:

- Usas **declarative_base()** para definir la clase base de los modelos que utilizarán SQLAlchemy.

Configuración de base de datos MongoDB:

- Si **MONGO_URL** está configurada, crea un cliente para MongoDB y define la base de datos y la colección específicas.
- Si **MONGO_URL** no está disponible, se emite una advertencia y se mantiene la conexión en None.

crud/ dispensaciones.py: Archivos para las operaciones de crear, leer, actualizar y eliminar (CRUD) sobre en todos los archivos del crud realizar las mismas funciones.

Importaciones:

- Session de **sqlalchemy.orm**: Para manejar las sesiones de la base de datos.
- Modelos y esquemas específicos para Dispensacion.

Funciones Definidas:

get_dispensacion(db: Session, dispensacion_id: int)

- Recupera una dispensación específica basada en su ID.
- Usa el método .first() para devolver el primer resultado o None si no se encuentra.

get_dispensaciones(db: Session, skip: int = 0, limit: int = 10)

- Recupera una lista de dispensaciones con soporte de paginación usando skip y limit.
- Devuelve todos los registros después de aplicar el desplazamiento y el límite.

create_dispensacion(db: Session, dispensacion: DispensacionCreate)

- Crea un nuevo registro de dispensación en la base de datos.
- Toma un objeto de tipo DispensacionCreate, convierte sus valores en un modelo de Dispensacion, y lo añade a la base de datos.
- Usa .commit() para guardar los cambios y .refresh() para cargar los datos actualizados.

update_dispensacion(db: Session, dispensacion_id: int, dispensacion: DispensacionUpdate)

- Busca una dispensación existente para actualizarla.
- Permite actualizar los campos que no sean None. Usa setattr para modificar los atributos de manera dinámica.
- Guarda los cambios después de la actualización.

delete_dispensacion(db: Session, dispensacion_id: int)

- Busca y elimina un registro de dispensación basado en su ID.
- Realiza una comprobación para asegurarse de que el registro exista antes de intentar eliminarlo.

Models/_init_.py: Permite la configuración adecuada de los modelos en SQLAlchemy.

Importaciones:

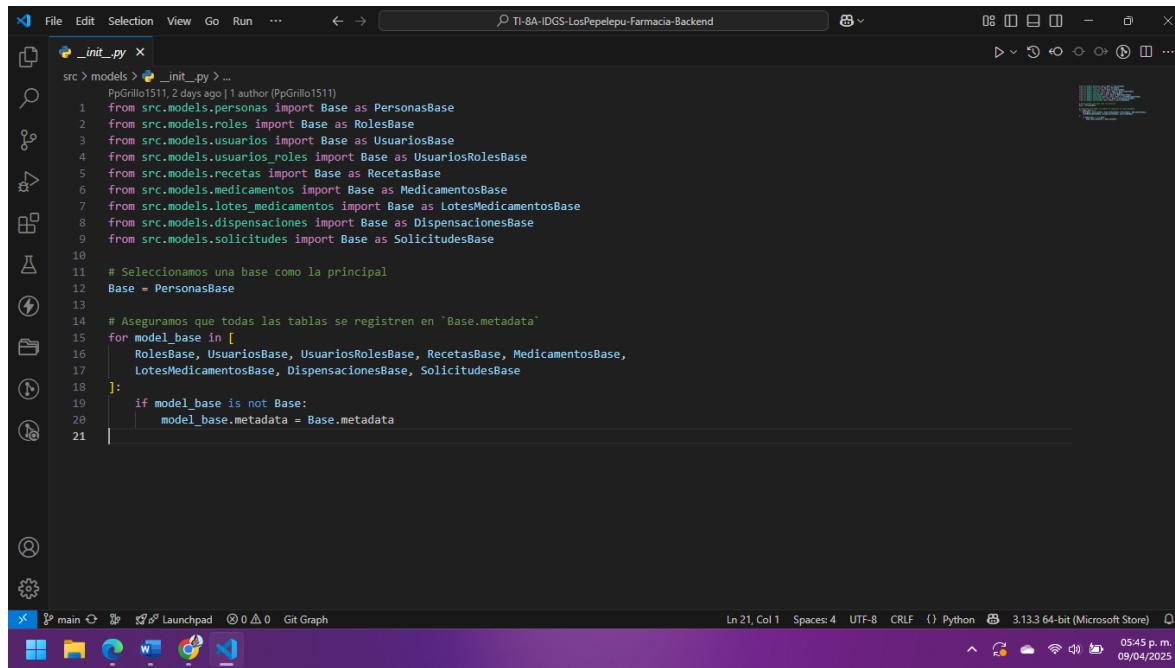
- Importas las clases base de los modelos de personas, roles, usuarios, usuarios_roles, recetas, medicamentos, lotes_medicamentos, dispensaciones, y solicitudes.

Definición de la Base:

- Seleccionas una de las bases (en este caso, PersonasBase) como la principal.

Registro de Tablas:

- Utilizas un bucle para asegurar que todos los modelos se registren en Base.metadata. Esto es importante para que SQLAlchemy reconozca las tablas al crear la base de datos o al utilizar operaciones ORM.
- Se evita que la base seleccionada se registre nuevamente al verificar que no sea igual a Base.



```

File Edit Selection View Go Run ... ← → ⌂ Ti-8A-IDGS-LosPepelepu-Farmacia-Backend ⌂
src > models > __init__.py > ...
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
1 from src.models.personas import Base as PersonasBase
2 from src.models.roles import Base as RolesBase
3 from src.models.usuarios import Base as UsuariosBase
4 from src.models.usuarios_roles import Base as UsuariosRolesBase
5 from src.models.recetas import Base as RecetasBase
6 from src.models.medicamentos import Base as MedicamentosBase
7 from src.models.lotes_medicamentos import Base as LotesMedicamentosBase
8 from src.models.dispensaciones import Base as DispensacionesBase
9 from src.models.solicitudes import Base as SolicitudesBase
10
11 # Seleccionamos una base como la principal
12 Base = PersonasBase
13
14 # Aseguramos que todas las tablas se registren en 'Base.metadata'
15 for model_base in [
16     RolesBase, UsuariosBase, UsuariosRolesBase, RecetasBase, MedicamentosBase,
17     LotesMedicamentosBase, DispensacionesBase, SolicitudesBase
18 ]:
19     if model_base is not Base:
20         model_base.metadata = Base.metadata
21

```

Models/dispensaciones.py

Importaciones:

- Importas las clases necesarias desde SQLAlchemy para definir las columnas y sus tipos, así como la relación entre modelos.
- Importas Base desde tu configuración de la base de datos.

Definición de la Clase:

- **__tablename__**: Define el nombre de la tabla en la base de datos como tbd_dispensaciones

Columnas:

- **ID**: Clave primaria que se incrementa automáticamente.
- **RecetaMedica_ID**: Clave foránea que hace referencia a tbd_recetas_medicas.ID. Es nullable para permitir que una dispensación no siempre esté asociada a una receta.
- **PersonalMedico_ID**: ID del personal médico, no puede ser nulo.
- **Departamento_ID**: ID del departamento, también no puede ser nulo.
- **Solicitud_ID**: ID de la solicitud, también nullable.
- **Estatus**: Enum que define el estado de la dispensación (abastecida o parcialmente abastecida).
- **Tipo**: Enum que define el tipo de dispensación (pública, privada, mixta).
- **TotalMedicamentosEntregados**: Número total de medicamentos entregados, no puede ser nulo.
- **Total_costo**: Costo total de la dispensación, no puede ser nulo.
- **Fecha_registro**: Fecha de registro, con un valor por defecto que se establece en el momento de la creación.
- **Fecha_actualizacion**: Fecha de última actualización, se actualiza automáticamente.

Relaciones:

- Definimos una relación con el modelo RecetaMedica, lo que permite acceder fácilmente a las dispensaciones asociadas a una receta a través de backref.

Models/lotes_medicamentos.py

Importaciones:

- Importas las clases necesarias desde SQLAlchemy para definir columnas y tipos de datos, así como datetime para gestionar fechas.
- Importas Base desde tu configuración de la base de datos.

Definición de la Clase:

- `__tablename__`: Define el nombre de la tabla en la base de datos como `tbd_lotes_medicamentos`.

Columnas:

- **ID**: Clave primaria que se incrementa automáticamente y se indexa para mejorar el rendimiento en las consultas.
- **Medicamento_ID**: ID del medicamento asociado (clave foránea que hace referencia a `tbc_medicamentos.ID`), no puede ser nulo.
- **PersonalMedico_ID**: ID del personal médico responsable, marcado como no nulo.
- **Clave**: Cadena de longitud máxima 50, no puede ser nula, representa una clave única para el lote.
- **Estatus**: Enum que define el estado del lote (reservado, en tránsito, recibido, rechazado), también no puede ser nulo.
- **Costo_Total**: Valor decimal para registrar el costo total del lote, con dos decimales.
- **Cantidad**: Cantidad de unidades en el lote, no puede ser nula.
- **Ubicacion**: Cadena de longitud máxima 100 que indica la ubicación del lote, no puede ser nula.
- **Fecha_registro**: Fecha y hora de creación, establecida por defecto utilizando `func.now()`.
- **Fecha_actualizacion**: Fecha de la última actualización, se actualizará automáticamente al modificar el registro.

Relaciones:

- `medicamento`: Se establece una relación con el modelo Medicamento. Esto permite acceso fácil a los datos del medicamento desde un objeto `LoteMedicamento`

Models/medicamentos.py

Importaciones:

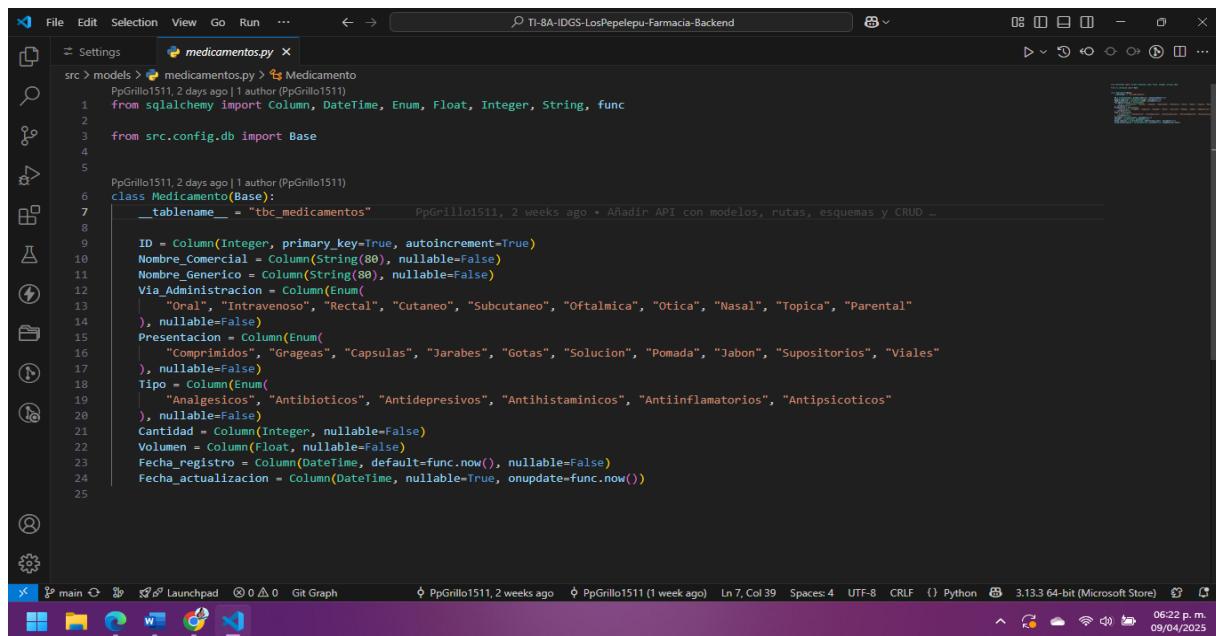
- Importas las clases necesarias desde SQLAlchemy para definir columnas y tipos de datos, así como func para gestionar fechas automáticamente.

Definición de la Clase:

- `__tablename__`: Nombrar la tabla en la base de datos como `tbc_medicamentos`.

Columnas:

- ID**: Clave primaria que se incrementa automáticamente.
- Nombre_Comercial**: Nombre comercial del medicamento, que es obligatorio (no puede ser nulo).
- Nombre_Generico**: Nombre genérico del medicamento, también obligatorio.
- Via_Administracion**: Enum que define la forma en que se administra el medicamento (varias opciones posibles).
- Presentacion**: Enum para definir la presentación del medicamento (varias formas disponibles).
- Tipo**: Enum que categoriza el medicamento (por tipo terapéutico).
- Cantidad**: Cantidad de unidades disponibles, que es un campo obligatorio.
- Volumen**: Indica el volumen del medicamento, también obligatorio.
- Fecha_registro**: Fecha de registro del medicamento, se establecerá automáticamente a la hora actual.
- Fecha_actualizacion**: Fecha de la última actualización, que se actualizará automáticamente al modificar el registro.



```

File Edit Selection View Go Run ... ← → ⌂ TI-8A-IDGS-LosPepepu-Farmacia-Backend
src > models > medicamentos.py > Medicamento
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
from sqlalchemy import Column, DateTime, Enum, Float, Integer, String, func
from src.config.db import Base

class Medicamento(Base):
    __tablename__ = "tbc_medicamentos"
    ID = Column(Integer, primary_key=True, autoincrement=True)
    Nombre_Comercial = Column(String(80), nullable=False)
    Nombre_Generico = Column(String(80), nullable=False)
    Via_Administracion = Column(Enum(
        "Oral", "Intravenoso", "Rectal", "Cutaneo", "Subcutaneo", "Oftalmica", "Otica", "Nasal", "Topica", "Parental"
    ), nullable=False)
    Presentacion = Column(Enum(
        "Comprimidos", "Grageas", "Capsulas", "Jarabes", "Gotas", "Solucion", "Pomada", "Jabon", "Supositorios", "Viales"
    ), nullable=False)
    Tipo = Column(Enum(
        "Analgesicos", "Antibioticos", "Antidepresivos", "Antihistaminicos", "Antiinflamatorios", "Antipsicoticos"
    ), nullable=False)
    Cantidad = Column(Integer, nullable=False)
    Volumen = Column(Float, nullable=False)
    Fecha_registro = Column(DateTime, default=func.now(), nullable=False)
    Fecha_actualizacion = Column(DateTime, nullable=True, onupdate=func.now())

```

Models/personas.py

Importaciones:

- Importas las librerías necesarias para definir caracteres y tipos que utilizarás en tu modelo, así como datetime para gestionar las fechas.
 - Importas Base desde tu configuración de la base de datos.

Definición de Enums:

- Definiste dos enums (GeneroEnum y GrupoSanguineoEnum), lo que ayuda a restringir los valores permitidos en las columnas correspondientes y hace que tu código sea más legible.

Definición de la Clase:

- tablename: Nombras la tabla en la base de datos como tbb_personas.

Columns:

- Usas Column para definir los atributos con tipos adecuados. Cada columna tiene una descripción dentro del comment, lo que facilita la comprensión del propósito de cada atributo.

Relaciones:

- Establece una relación con el modelo Usuario, lo que te permitirá acceder a los datos del usuario asociado desde el objeto Persona.

Models/recetas.py

- **Columnas:** Cada columna tiene un comentario que detalla su propósito, lo que hace más fácil entender el modelo sin necesidad de una documentación externa.
 - **Tipos de Datos:** Se utilizan varios tipos de datos para reflejar adecuadamente la información que se almacena en cada columna.
 - **Registro de Fechas:** Las columnas para las fechas manejan registro automático al crearse y actualizarse, lo que asegura que la información de auditoría esté siempre disponible.
 - **Relaciones:** Si este modelo tiene alguna relación con otros modelos, podrías agregarlas como lo has hecho con los otros modelos.

```
File Edit Selection View Go Run ... ← → Ti-8A-IDGS-LosPepelepu-Farmacia-Backend 08 □ □ - ... Explorer Settings src > models > recetas.py < recetas.py > ... PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
1  from sqlalchemy import Column, DateTime, Integer, String, Text, func
2  from sqlalchemy.orm import relationship
3
4  from src.config.db import Base
5
6
7  PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
8  class RecetaMedica(Base):
9      __tablename__ = "tbl_recetas_medicas"
10
11      ID = Column(Integer, primary_key=True, autoincrement=True)
12      Paciente_Nombre = Column(String(100), nullable=False)
13      Paciente_Edad = Column(Integer, nullable=False)
14      Medico_Nombre = Column(String(100), nullable=False)
15      Fecha = Column(DateTime, default=func.now(), nullable=False)
16      Diagnostico = Column(String(255), nullable=True)
17      Medicamentos = Column(Text, nullable=True)
18      Indicaciones = Column(Text, nullable=True)
19      Fecha_registro = Column(DateTime, default=func.now(), nullable=False)
20      Fecha_actualizacion = column(DateTime, nullable=True, onupdate=func.now())
```

Models/roles.py

Importaciones:

- Importas las librerías necesarias para definir columnas y tipos que utilizarás en tu modelo, así como datetime para gestionar las fechas.
 - Importas Base desde tu configuración de la base de datos.

Definición de la Clase:

- tablename: Nombras la tabla en la base de datos como tbc_roles.

Columnas:

- Usas Column para definir los atributos con tipos adecuados. Cada columna tiene una descripción dentro del comment, lo que facilita la comprensión del propósito de cada atributo:
 - **ID**: Identificador único del rol.
 - **Nombre**: Nombre del rol dentro del sistema.
 - **Descripcion**: Información detallada sobre funciones y permisos del rol.
 - **Estatus**: Estado actual del rol (activo o inactivo).
 - **Fecha_Registro**: Fecha y hora en las que se registró el rol.
 - **Fecha_Actualizacion**: Fecha y hora de la última actualización del rol.

Relaciones:

- Establece una relación con UsuarioRol, lo que te permitirá acceder a los usuarios asociados a cada rol.

```
roles.py x
src > models > roles.py > ...
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
1 from datetime import datetime
2
3 from sqlalchemy import Boolean, Column, DateTime, Integer, String, Text
4 from sqlalchemy.ext.declarative import declarative_base
5 from sqlalchemy.orm import relationship
6
7 from src.config.db import Base
8
9
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
10 class Role(Base):
11     __tablename__ = "tbc_roles"
12
13     ID = Column(Integer, primary_key=True, autoincrement=True, comment="")
14         Descripción: Identificador único del rol
15         Naturaleza: Cualitativo.
16         Dominio: Caracteres hexadecimales (0-F).
17         Composición: 8(0-F)-4(0-F)-4(0-F)-4(0-F)-12(0-F)
18     """
19
20     Nombre = Column(String(50), nullable=False, comment="")
21         Descripción: Nombre del rol dentro del sistema.
22         Naturaleza: Cualitativo.
23         Dominio: Caracteres alfabéticos.
24         Composición: Entre 1 y 50 caracteres (a-z | A-Z).
25     """
26
27     Descripcion = Column(Text, nullable=True, comment="")
28         Descripción: Información detallada sobre las funciones y permisos del rol.
29         Naturaleza: Cualitativo.
30         Dominio: Caracteres alfanuméricos y símbolos.
```

Models/solicitudes.py

Importaciones:

- Importas las librerías necesarias para definir tipos de columnas en SQLAlchemy y gestionar la datetime.
- Importas Base desde tu configuración de la base de datos.

Definición de la Clase:

- `__tablename__`: Nombras la tabla en la base de datos como `tbd_solicitudes`.

Columnas:

- **ID**: Identificador único de cada solicitud.
- **Paciente_ID**: Identificador del paciente asociado a la solicitud (actualmente es solo un entero sin clave foránea).
- **Medico_ID**: Identificador del médico que maneja la solicitud (también solo un entero).
- **Servicio_ID**: Identificador del servicio relacionado con la solicitud.
- **Prioridad**: Enum que categoriza la urgencia de la solicitud con valores predefinidos.
- **Descripcion**: Texto que contiene detalles sobre la solicitud, requisito importante para la interpretación de la misma.
- **Estatus**: Enum que define el estado actual de la solicitud, con un valor por defecto de "Registrada".
- **Estatus_Aprobacion**: Booleano que indica si la solicitud ha sido aprobada o no, con un valor por defecto de False.
- **Fecha_Registro**: Fecha y hora en que se registró la solicitud, utilizando UTC.
- **Fecha_Actualizacion**: Fecha y hora de la última actualización de la solicitud.

Relaciones:

- Las relaciones con Paciente, Medico y Servicio, implementar estas relaciones para poder navegar entre los distintos modelos.

Models/usuarios_roles.py

Importaciones:

- Importas las librerías adecuadas de SQLAlchemy para definir tipos de columnas, gestionar la datetime y establecer relaciones entre tablas.
- Importas Base desde tu configuración de la base de datos.

Definición de la Clase:

- `__tablename__`: Nombras la tabla en la base de datos como `tbd_usuarios_roles`.

Columnas:

- **Usuario_ID**: Identificador único del usuario, que es también una clave foránea que referencia a `tbb_usuarios`. Se define como `nullable=False`, lo que asegura que cada relación debe tener un usuario asociado.
- **Rol_ID**: Identificador único del rol asignado al usuario, con una referencia a `tbc_roles`. También `nullable=False` para garantizar la integridad de la relación.
- **Estatus**: Estado de la relación entre el usuario y su rol, definido como un booleano con un valor por defecto de `True` (activo).
- **Fecha_Registro**: Se registra automáticamente la fecha y hora en que se creó la relación entre el usuario y el rol, usando `datetime.utcnow`.
- **Fecha_Actualizacion**: Se actualiza automáticamente en caso de modificaciones, manteniendo un registro de la última vez que se modificó la relación.

Relaciones:

- Se establecen relaciones many-to-one con `Usuario` y `Role`, utilizando `back_populates` para facilitar la navegación entre estos modelos y establecer una relación bidireccional.

Models/usuarios.py

Importaciones:

- Importas las librerías necesarias, incluyendo datetime para gestionar las fechas y la clase enum para definir los estados del usuario.
- Importas Base desde tu configuración de la base de datos.

Definición de la Clase:

- **__tablename__**: Nombras la tabla en la base de datos como tbb_usuarios.

Columnas:

- **ID**: Identificador único autoincrementable del usuario.
- **Persona_ID**: Clave foránea que referencia a la tabla tbb_personas, asegurando que cada usuario esté asociado a una persona válida.
- **Nombre_Usuario**: Cadenas alfanuméricas únicas que usarán los usuarios para acceder al sistema.
- **Correo_Electronico**: Correo electrónico único asociado a cada usuario, lo que es fundamental para la gestión de cuentas y notificaciones.
- **Contrasena**: Contraseña cifrada, un aspecto crítico para la seguridad del acceso.
- **Estatus**: Enum que indica el estado del usuario, con opciones como Activo, Inactivo, Bloqueado, y Suspendido.
- **Fecha_Registro**: Fecha y hora de creación de la cuenta, almacenada en UTC.
- **Fecha_Actualizacion**: Registra la fecha y hora de la última modificación de los datos del usuario.

Relaciones:

Estableces una relación uno a uno con la clase Persona, permitiendo acceder a los datos de la persona asociada.

También estableces una relación uno a muchos con UsuarioRol, que permite gestionar los roles del usuario en el sistema.

Routes/dispensaciones.py

Importaciones:

- Importas las librerías necesarias de FastAPI, SQLAlchemy, y definiciones personalizadas para el manejo de la base de datos, CRUD y esquemas.
- El tipo List se importa para manejar listas en las respuestas.

Definición del Router:

- Creas un APIRouter llamado dispensacion_router para manejar las rutas relacionadas con dispensaciones.

Dependencia de la Base de Datos:

- La función get_db se utiliza para gestionar la sesión de base de datos, asegurando que se cierre adecuadamente después de su uso.

Rutas Definidas:

Leer todas las dispensaciones:

- Método GET en /dispensaciones/, con soporte para paginación usando skip y limit.
- Responde con una lista de objetos de tipo Dispensacion.

Leer una dispensación específica:

- Método GET en /dispensacion/{id} que busca una dispensación por su ID. Lanza una excepción 404 si no se encuentra.

Crear una nueva dispensación:

- Método POST en /dispensaciones/, aceptando un objeto de tipo DispensacionCreate.

Actualizar una dispensación existente:

- Método PUT en /dispensacion/{id} que actualiza una dispensación específica y lanza una excepción 404 si no se encuentra.

Eliminar una dispensación:

- Método DELETE en /dispensacion/{id} que elimina una dispensación específica, lanzando una excepción 404 si no se encuentra.

Routes/lotes_medicamentos.py

Importaciones:

- Importas las librerías necesarias de FastAPI, así como las funciones CRUD de lotes de medicamentos y los esquemas correspondientes.
- La tipificación es clara, utilizando List y Optional para definir la estructura de datos esperada.

Definición del Router:

- Creas un APIRouter llamado lotes_medicamentos_router con un prefijo definido y etiquetas para la documentación de la API.

Dependencia de la Base de Datos:

- Defines get_db() como una función asíncrona que maneja la conexión a la base de datos. Sin embargo, SQLAlchemy no está diseñado para funcionar de manera asíncrona de manera predeterminada. Aunque utilizas async def, deberías tener en cuenta esta limitación, y es recomendable utilizar un marco que soporte operaciones asíncronas o dejar los métodos como síncronos.

Rutas Definidas:

- **Crear nuevo lote:** Método POST en "/" que crea un nuevo lote utilizando datos de LoteMedicamentoCreate.
- **Obtener un lote específico:** Método GET en "{lote_id}" que busca un lote por su ID y lanza una excepción 404 si no se encuentra.
- **Obtener todos los lotes (paginados):** Método GET en "/" que permite la paginación mediante los parámetros skip y limit.
- **Actualizar un lote existente:** Método PUT en "{lote_id}" para actualizar un lote existente, lanzando una excepción 404 si no se encuentra.
- **Eliminar un lote:** Método DELETE en "{lote_id}" que elimina un lote específico, lanzando también una excepción 404 si no se encuentra.

Routes/medicamentos.py

Importaciones:

- Importas las librerías necesarias, incluyendo FastAPI y SQLAlchemy. Está bien organizado, y utilizas dependencias que facilitan la inyección de sesiones de base de datos y la autenticación.

Definición del Router:

- Creas un APIRouter llamado medicamento_router, que tiene un propósito claro para gestionar los medicamentos.

Gestión de la Base de Datos:

- La función get_db es responsable de abrir y cerrar la sesión de la base de datos correctamente, lo cual es crucial para evitar pérdidas de conexiones.

Rutas Definidas:

- **Leer todos los medicamentos:** **GET /medicamentos/**: Permite la paginación utilizando los parámetros skip y limit.
- **Leer un medicamento específico:** **GET /medicamento/{ID}**: Recupera un medicamento por su ID, con manejo de errores si no se encuentra.
- **Crear un nuevo medicamento:** **POST /medicamentos/**: Crea un nuevo medicamento.
- **Actualizar un medicamento existente:** **PUT /medicamento/{ID}**: Actualiza un medicamento y maneja el error si no existe.
- **Eliminar un medicamento:** **DELETE /medicamento/{ID}**: Elimina un medicamento, lanzando un error 404 si no se encuentra.

Routes/personas.py

Importaciones:

- Estás importando las dependencias necesarias de FastAPI y SQLAlchemy, así como las funciones CRUD y los esquemas relacionados con personas.
- La estructura de importación es limpia y fácil de seguir.

Definición del Router:

- Creas un APIRouter llamado personas_router, con un prefijo "/personas" y etiquetas correspondientes para organizar la documentación de tu API.

Gestión de la Base de Datos:

- La función get_db() maneja la sesión de la base de datos, abriendo y cerrando la conexión adecuadamente.

Rutas Definidas:

- **Crear una nueva persona:** **POST /personas/**: Permite crear una nueva persona y espera un objeto de tipo PersonaCreate.
- **Leer una persona específica:** **GET /{persona_id}**: Recupera una persona por su ID, manejando el caso en el que la persona no se encuentra.
- **Leer todas las personas:** **GET /**: Devuelve una lista de personas, con soporte para paginación a través de los parámetros skip y limit.
- **Actualizar una persona existente:** **PUT /{persona_id}**: Permite actualizar los datos de una persona, lanzando una excepción 404 si no se encuentra.
- **Eliminar una persona:** **DELETE /{persona_id}**: Elimina una persona por su ID y lanza un error si no se encuentra.

Routes/recetas.py

Importaciones:

- Has importado los módulos necesarios de FastAPI, SQLAlchemy y tus componentes personalizados (CRUD y esquemas).
- La organización de los imports es clara y fácil de seguir.

Definición del Router:

- Creas un APIRouter llamado receta_router, que tiene como objetivo organizar las rutas relacionadas con las recetas médicas.

Gestión de la Base de Datos:

- La función get_db() maneja la conexión a la base de datos, asegurando que la sesión se cierre correctamente después de su uso. Esto es fundamental para evitar fugas de conexión.

Rutas Definidas:

- **Leer todas las recetas:** `GET /recetas/`: Permite la paginación a través de los parámetros skip y limit.
- **Leer una receta específica:** `GET /receta/{id}`: Busca una receta por su ID, manejando el error 404 si no se encuentra.
- **Crear una nueva receta:** `POST /recetas/`: Permite crear una nueva receta y retorna el objeto creado.
- **Actualizar una receta existente:** `PUT /receta/{id}`: Actualiza una receta existente y lanza un error 404 si no cumple con los requisitos.
- **Eliminar una receta:** `DELETE /receta/{id}`: Elimina una receta y lanza un error 404 si no se encuentra.

Routes/solicitudes.py

Importaciones:

- Estás importando las dependencias clave de FastAPI y SQLAlchemy, así como las funciones CRUD específicas para manejar las solicitudes y los esquemas correspondientes.
- La importación de Portador para las dependencias de autenticación es efectiva para proteger tus rutas.

Definición del Router:

- Has creado un APIRouter llamado solicitudes_router, que incluye un prefijo /solicitudes, lo cual es un enfoque adecuado para agrupar rutas relacionadas.

Gestión de la Base de Datos:

- La función get_db() gestiona de forma correcta la sesión de la base de datos, asegurando que se cierre al final de su uso.

Rutas Definidas:

- **Crear una nueva solicitud: POST /solicitudes/**: Permite la creación de una nueva solicitud, retornando el objeto creado.
- **Leer una solicitud específica: GET /{solicitud_id}**: Permite obtener una solicitud por su ID. Manejas el caso en el que no se encuentra la solicitud de forma correcta.
- **Leer todas las solicitudes: GET /**: Devuelve una lista de solicitudes con soporte para paginación a través de parámetros skip y limit.
- **Actualizar una solicitud existente: PUT /{solicitud_id}**: Permite actualizar una solicitud existente y maneja el caso en que la solicitud no se encuentra.
- **Eliminar una solicitud: DELETE /{solicitud_id}**: Elimina la solicitud con el ID especificado y maneja el caso de que no se encuentre.

Routes/usuarios_roles.py

Importaciones:

- Has importado las dependencias necesarias de FastAPI, SQLAlchemy y tus componentes personalizados (CRUD y esquemas).
- La importación de Portador para la autenticación es bien manejada y permite proteger algunas rutas.

Definición del Router:

- Has creado un APIRouter llamado usuarios_roles_router con un prefijo /usuarios-roles, lo que organiza claramente las rutas relacionadas.

Gestión de la Base de Datos:

- La función get_db() gestiona la conexión a la base de datos correctamente, asegurando que la sesión se cierre al final de su uso.

Rutas Definidas:

- **Crear un nuevo usuario-rol: POST /usuarios-roles/**: Permite la creación de una nueva asignación de usuario-rol. Esta ruta no está protegida, lo que podría ser un vector de riesgo.
- **Leer una asignación específica de usuario-rol: GET /{usuario_rol_id}**: Permite obtener un usuario-rol por su ID, con manejo de errores si no se encuentra.
- **Leer todos los usuarios-roles: GET /**: Obtiene una lista de usuario-roles con soporte para paginación.
- **Actualizar una asignación de usuario-rol existente: PUT /{usuario_rol_id}**: Actualiza un usuario-rol existente, manejando el caso donde no se encuentra.
- **Eliminar una asignación de usuario-rol: DELETE /{usuario_rol_id}**: Elimina un usuario-rol específico, lanzando un error si no se encuentra.

Routes/usuarios.py

Estructura del Router:

- Has creado un APIRouter llamado usuarios_router con un prefijo /usuarios, que es una buena práctica para mantener las rutas organizadas.

Manejo de la Base de Datos:

- La función get_db() gestiona eficientemente las conexiones a la base de datos, asegurando que las sesiones se cierren adecuadamente.

Rutas Definidas:

- **Crear un nuevo usuario:** (POST /usuarios/): Esta ruta permite crear un nuevo usuario y no requiere autenticación.
- **Obtener un usuario por ID:** (GET /usuarios/{usuario_id}): Esta ruta requiere autenticación y devuelve un usuario específico por su ID.
- **Obtener todos los usuarios:** (GET /usuarios/): También requiere autenticación y permite la paginación.
- **Actualizar un usuario:** (PUT /usuarios/{usuario_id}): Requiere autenticación y actualiza un usuario existente.
- **Eliminar un usuario:** (DELETE /usuarios/{usuario_id}): Requiere autenticación para eliminar un usuario.
- **Login de un usuario:** (POST /usuarios/login): Permite a un usuario autenticarse sin necesidad de estar autenticado previamente.

schemas/_init_.py.py

Importación de Bases de Modelos:

- Estás importando las clases base (Base) de diferentes modelos. Esto es necesario para que SQLAlchemy pueda gestionar las tablas en la base de datos.

Definición de la Base Principal:

- Has seleccionado PersonasBase como la base principal. Esto está bien si PersonasBase es la clase que contiene la mayoría de tus modelos relacionados. Sin embargo, si la mayoría de tus modelos no pertenecen a PersonasBase, considera seleccionar la base que contenga más modelos.

Registro de Metadatos:

- El ciclo for que tienes registra todas las tablas para las clases base diferentes en Base.metadata. Esto garantiza que todos los modelos estén conectados al mismo conjunto de metadatos y, por lo tanto, se puedan crear y administrar correctamente.

schemas/dispensaciones.py

BaseModel:

Has creado una clase base DispensacionBase que incluye los campos comunes para las operaciones CRUD. Esta es una buena práctica para evitar la duplicación de código.

Uso de Tipos Opcionales:

Utilizas Optional para algunos campos que pueden no estar presentes. Esto es correcto y proporciona flexibilidad en el momento de la creación o actualización de una dispensación.

DateTime:

Los campos de tipo datetime están bien definidos. Sin embargo, si quieras restringir el formato de las fechas en la entrada, podrías considerar usar validaciones adicionales.

Clases Derivadas:

Has definido clases DispensacionCreate, DispensacionUpdate y Dispensacion, lo que permite extender la funcionalidad sin modificar la clase base.

Configuración adicional:

La clase Config dentro de la clase Dispensacion es útil para establecer configuraciones adicionales de Pydantic, como permitir la creación de atributos de forma dinámica.

schemas/lotes_medicamentos.py

BaseModel:

Hemos creado una clase base llamada LoteMedicamentoBase, en la que definimos todos los campos comunes que se utilizan para las operaciones CRUD (crear, leer, actualizar y eliminar).

Esto nos permite mantener un código limpio y evitar duplicaciones innecesarias.

```
class LoteMedicamentoBase(BaseModel):
    Medicamento_ID: int
    PersonalMedico_ID: int
    Clave: str
    Estatus: str
    Costo_Total: float
    Cantidad: int
    Ubicacion: str
```

Uso de Tipos Opcionales:

- Realizamos una clase LoteMedicamentoUpdate en la que usamos Optional para permitir que los campos sean enviados de forma parcial durante una actualización.
- Esta flexibilidad es ideal para escenarios donde no todos los valores necesitan ser modificados.

```
class LoteMedicamentoUpdate(BaseModel):
    Estatus: Optional[str] = None
    Costo_Total: Optional[float] = None
    Cantidad: Optional[int] = None
    Ubicacion: Optional[str] = None
    Fecha_actualizacion: Optional[datetime] = None
```

Manejo de Fechas (DateTime):

- Incluimos campos de tipo datetime para llevar un control detallado sobre los momentos de registro y de modificación del lote.
- Esto nos ayuda a tener una trazabilidad clara del historial de cada registro.

Clases Derivadas:

Dividimos el modelo en varias clases para manejar distintos casos de uso:

- LoteMedicamentoCreate: Se basa en la clase base y la utilizamos al momento de crear un nuevo lote.
- LoteMedicamentoUpdate: Define únicamente los campos que pueden modificarse, y todos son opcionales.
- LoteMedicamentoInDBBase: Representa un modelo completo desde la base de datos, incluyendo el identificador (ID) y las fechas.
- LoteMedicamento: Es la clase final que usamos para retornar la información completa al cliente.

Configuración adicional (Config):

- Incluimos una clase Config dentro del modelo LoteMedicamentoInDBBase, en la que activamos la opción `orm_mode = True`.
- Esto permite que los objetos ORM, como los de SQLAlchemy, puedan ser convertidos directamente a modelos Pydantic sin necesidad de pasos intermedios.

```
class Config:  
    orm_mode = True # Asegura que los modelos se puedan convertir correctamente desde SQLAlchemy
```

schemas/ medicamentos:

Hemos creado la clase MedicamentoBase como estructura principal que agrupa todos los campos comunes relacionados con los medicamentos.

Esta clase nos permite centralizar la lógica base y reutilizarla fácilmente en operaciones como creación, actualización y respuesta desde la base de datos..py

```
class MedicamentoBase(BaseModel):
    Nombre_Comercial: str
    Nombre_Generico: str
    Via_Administracion: str
    Presentacion: str | PpGrillo1511, 2 weeks ago
    Tipo: str
    Cantidad: int
    Volumen: float
    Fecha_registro: datetime
    Fecha_actualizacion: Optional[datetime] = None
```

Creación de Registros:

Definimos la clase MedicamentoCreate, que hereda directamente de MedicamentoBase.

La utilizamos al momento de registrar un nuevo medicamento, aprovechando todos los campos definidos en la base.

```
class MedicamentoCreate(MedicamentoBase):
    pass
```

Actualización de Registros:

Creamos la clase MedicamentoUpdate, también basada en MedicamentoBase.

Actualmente, permite actualizar todos los campos, aunque si se desea hacer que solo algunos sean opcionales, podríamos redefinirla con Optional.

```
class MedicamentoUpdate(MedicamentoBase):
    pass
```

Respuesta con Identificador:

Incluimos la clase Medicamento, que extiende MedicamentoBase y agrega el campo ID.

Este modelo representa un objeto completo que incluye la información que devolvemos desde la base de datos al cliente.

```
class Medicamento(MedicamentoBase):
    ID: int

PpGrillo1511, 2 weeks ago | 1 author (PpGrillo1511)
    class Config:
        from_attributes = True
```

Configuración adicional (Config):

Agregamos una clase interna Config con la opción from_attributes = True. Esto es equivalente a orm_mode = True, y permite que los modelos funcionen correctamente con objetos ORM como los de SQLAlchemy.

schemas/ personas.py:

Uso de Enumeraciones (Enum):

Hemos definido dos enumeraciones que nos permiten restringir los valores aceptados para ciertos campos, asegurando consistencia en los datos.

Usamos GeneroEnum para limitar los géneros permitidos a Masculino, Femenino y No Binario.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class GeneroEnum(str, Enum):
    M = "M"
    F = "F"
    NB = "NB"
```

GrupoSanguíneoEnum restringe los valores a los tipos sanguíneos estándar, evitando errores de ingresos.

```
class GrupoSanguíneoEnum(str, Enum):
    A_POS = "A+"
    A_NEG = "A-"
    B_POS = "B+"
    B_NEG = "B-"
    AB_POS = "AB+"
    AB_NEG = "AB-"
    O_POS = "O+"
    O_NEG = "O-"
```

BaseModel:

Creamos PersonaBase como clase base con todos los campos esenciales que representan a una persona.

Esto nos permite reutilizar esta definición en las operaciones de creación, actualización y respuesta desde la base de datos.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class PersonaBase(BaseModel):
    Titulo: Optional[str] = None
    Nombre: str
    Primer_Apellido: str
    Segundo_Apellido: Optional[str] = None
    CURP: Optional[str] = None
    Genero: GeneroEnum
    Grupo_Sanguíneo: GrupoSanguíneoEnum
    Fecha_Nacimiento: date
    Estatus: bool = True
```

Creación de Registros:

Definimos PersonaCreate como clase heredada de PersonaBase.

La usamos cuando queremos registrar una nueva persona.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class PersonaCreate(PersonaBase):
    pass
```

Actualización de Registros:

Creamos PersonaUpdate, donde todos los campos son opcionales.

Esto nos permite realizar actualizaciones parciales (por ejemplo, mediante una solicitud tipo PATCH).

```
class PersonaUpdate(BaseModel):
    Titulo: Optional[str] = None
    Nombre: Optional[str] = None
    Primer_Apellido: Optional[str] = None
    Segundo_Apellido: Optional[str] = None
    CURP: Optional[str] = None
    Genero: Optional[GeneroEnum] = None
    Grupo_Sanguineo: Optional[GrupoSanguineoEnum] = None
    Fecha_Nacimiento: Optional[date] = None
    Estatus: Optional[bool] = None
```

Modelo para la base de datos:

Incluimos la clase PersonalInDBBase, que hereda de PersonaBase y agrega campos importantes para control de registro, como el ID, Fecha_Registro y Fecha_Actualizacion.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class PersonaInDBBase(PersonaBase):
    ID: int
    Fecha_Registro: datetime
    Fecha_Actualizacion: Optional[datetime] = None

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class Config:
    from_attributes = True
```

schemas/recetas.py:

Modelo Base (RecetaBase)

- Incluye los campos esenciales de una receta médica:
- Paciente_Nombre, Paciente_Edad, Medico_Nombre, Fecha
- Campos opcionales como: Diagnostico, Medicamentos, Indicaciones, Fecha_registro.

```
class RecetaBase(BaseModel):
    Paciente_Nombre: str
    Paciente_Edad: int
    Medico_Nombre: str
    Fecha: datetime
    Diagnostico: Optional[str] = None
    Medicamentos: Optional[str] = None
    Indicaciones: Optional[str] = None
    Fecha_registro: Optional[datetime] = None
```

Modelos Específicos

- RecetaCreate: hereda directamente de RecetaBase
- RecetaUpdate: también hereda igual, pero puede mejorarse haciendo los campos opcionales si se desea actualización parcial

```
PpGrillo1511, 2 weeks ago | 1 author (PpGrillo1511)
class RecetaCreate(RecetaBase):
    pass

PpGrillo1511, 2 weeks ago | 1 author (PpGrillo1511)
class RecetaUpdate(RecetaBase):
    pass
```

Modelo para respuesta (Receta)

- Incluye el campo ID y usa configuración compatible con ORMs:

```
PpGrillo1511, 2 weeks ago | 1 author (PpGrillo1511)
class RecetaCreate(RecetaBase):
    pass
```

schemas/ solicitudes.py:

Modelos Específicos

- SolicitudCreate: hereda directamente de SolicitudBase
- SolicitudUpdate: también hereda, pero permite campos opcionales para una actualización parcial.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class SolicitudBase(BaseModel):
    Paciente_ID: int
    Medico_ID: int
    Servicio_ID: int
    Prioridad: str
    Descripcion: str
    Estatus: Optional[str] = "Registrada"
    Estatus_Aprobacion: Optional[bool] = False
```

Modelos Específicos

- SolicitudCreate: hereda directamente de SolicitudBase
- SolicitudUpdate: también hereda, pero permite campos opcionales para una actualización parcial.

```
PpGrillo1511, 2 weeks ago | 1 author (PpGrillo1511)
class SolicitudCreate(SolicitudBase):
    pass

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class SolicitudUpdate(BaseModel):
    Prioridad: Optional[str]
    Descripcion: Optional[str]
    Estatus: Optional[str]
    Estatus_Aprobacion: Optional[bool]
    Fecha_Actualizacion: Optional[datetime] = Field(default_factory=datetime.utcnow)
```

Modelo para respuesta (Solicitud)

- Incluye el campo ID y usa configuración compatible con ORMs:

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class SolicitudInDBBase(SolicitudBase):
    ID: int
    Fecha_Registro: datetime
    Fecha_Actualizacion: Optional[datetime]

PpGrillo1511, 2 weeks ago | 1 author (PpGrillo1511)
class Config:
    from_attributes = True
```

schemas/usuarios_roles.py:

Modelo Base (UsuarioRolBase):

Incluye los campos esenciales de un rol de usuario:

- Usuario_ID, Rol_ID, Estatus, Fecha_Registro

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioRolBase(BaseModel):
    Usuario_ID: int
    Rol_ID: int
    Estatus: bool = True
    Fecha_Registro: datetime
```

Modelos Específicos

- UsuarioRolCreate: hereda directamente de UsuarioRolBase.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioRolCreate(UsuarioRolBase):
    pass
```

- UsuarioRolUpdate: permite campos opcionales para una actualización parcial.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioRolUpdate(BaseModel):
    Estatus: Optional[bool] = None
    Fecha_Actualizacion: Optional[datetime] = None

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class Config:
    from_attributes = True
```

Modelo para respuesta (UsuarioRol)

- Incluye el campo Fecha_Actualizacion y usa configuración compatible con ORMs:

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioRolInDBBase(UsuarioRolBase):      PpGrill
    Fecha_Actualizacion: Optional[datetime] = None

    class Config:
        from_attributes: bool
        from_attributes = True

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioRol(UsuarioRolInDBBase):
    pass
```

schemas/usuarios.py:

Modelo Base (UsuarioBase)

Incluye los campos esenciales de un usuario:

- Persona_ID, Nombre_Usuario, Correo_Electronico, Contrasena, Estatus, Fecha_Registro.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioBase(BaseModel):
    Persona_ID: int
    Nombre_Usuario: str
    Correo_Electronico: str
    Contrasena: str
    Estatus: EstatusUsuario
    Fecha_Registro: datetime
```

numeración de Estatus (EstatusUsuario)

- Define los posibles estados de un usuario:

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class EstatusUsuario(str, Enum):
    Activo = "Activo"
    Inactivo = "Inactivo"
    Bloqueado = "Bloqueado"
    Suspendido = "Suspendido"
```

Modelos Específicos

- UsuarioCreate: hereda directamente de UsuarioBase.

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioCreate(UsuarioBase):
    pass
```

UsuarioUpdate: permite campos opcionales para una actualización parcial

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioUpdate(BaseModel):
    Nombre_Usuario: Optional[str] = None
    Correo_Electronico: Optional[str] = None
    Contrasena: Optional[str] = None
    Estatus: Optional[EstatusUsuario] = None
    Fecha_Actualizacion: Optional[datetime] = None

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class Config:
    from_attributes = True
```

Modelo para respuesta (Usuario)

- Incluye el campo ID y Fecha_Actualizacion, y usa configuración compatible con ORMs:

```
PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class UsuarioInDBBase(UsuarioBase):          PpGrillo1511, 2 days ago
    ID: int
    Fecha_Actualizacion: Optional[datetime] = None

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class Config:
    from_attributes = True

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)
class Usuario(UsuarioInDBBase):
    pass
```

Modelo de Login (UsuarioLogin)

- Define los campos necesarios para el inicio de sesión:

```
PpGrillo1511, yesterday | 1 author (PpGrillo1511)
class UsuarioLogin(BaseModel):
    Nombre_Usuario: str
    Contrasena: str          PpGrillo1511, yesterday
```

security/ portadortoken.py:

Función get_db

- Propósito: Proporciona una sesión de base de datos utilizando un generador.
- Uso: Se cierra automáticamente después de su uso.

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

Clase Portador (HTTPBearer)

- Propósito: Extiende la funcionalidad de HTTPBearer para autenticar usuarios utilizando un token JWT.

Método call

Flujo:

- Obtiene el token de autorización del encabezado de la solicitud.
- Decodifica y valida el token utilizando valida_token.
- Extrae los campos necesarios del payload del token (user_name, email, phone_number, password).
- Valida que todos los campos necesarios estén presentes. Si falta alguno, lanza un HTTPException con un código de estado 400.
- Busca al usuario en la base de datos usando los datos obtenidos del token.
- Si el usuario no existe o los datos no coinciden, lanza un HTTPException con un código de estado 404.
- Maneja errores de decodificación del token y otros errores genéricos, lanzando un HTTPException con un código de estado 401.

PpGrillo1511, 2 days ago | 1 author (PpGrillo1511)

```

class Portador(HTTPBearer):
    async def call(self, request: Request, db: Session = Depends(get_db)):
        # Obtener el token de autorización
        autorizacion = await super().call(request)

        try:
            # Decodificar y validar el token
            dato = valida_token(autorizacion.credentials)

            # Validar que los campos necesarios están presentes en el payload
            user_name = dato.get("user_name")
            email = dato.get("email")
            phone_number = dato.get("phone_number")
            password = dato.get("password")

            if not all([user_name, email, phone_number, password]):
                raise HTTPException(status_code=400, detail="Token mal formado. Faltan datos de usuario.")

            # Buscar al usuario en la Base de datos con los datos obtenidos del token
            db_userlogin = src.crud.usuarios.get_user_by_credentials(
                db, username=user_name, correo=email, telefono=phone_number, password=password)

            # Si el usuario no existe o los datos no coinciden, lanzar error
            if db_userlogin is None:
                raise HTTPException(status_code=404, detail="Login incorrecto")

        return db_userlogin
  
```

Seeders/rolesSeeder.py

Función seed_roles

- Propósito: Inserta roles predeterminados en la base de datos.

Parámetros

- db (Session): Sesión de base de datos activa utilizada para realizar las operaciones de inserción.

Flujo

Definición de Roles:

- Se define una lista de instancias de Role, cada una con un Nombre y Descripción.

```
def seed_roles(db: Session):
    roles = [
        Role(Nombre="Administrador", Descripcion="Usuario con permisos completos para gestionar el sistema."),
        Role(Nombre="Farmacéutico", Descripcion="Responsable de la dispensación y control de medicamentos."),
        Role(Nombre="Cajero", Descripcion="Maneja las transacciones de ventas y cobros en la farmacia."),
        Role(Nombre="Médico", Descripcion="Encargado de realizar consultas y emitir recetas médicas."),
        Role(Nombre="Auxiliar", Descripcion="Brinda apoyo en el inventario y atención al cliente."),
        Role(Nombre="Cliente", Descripcion="Usuario normal")
    ]
```

Inserción en la Base de Datos:

- Se itera sobre la lista de roles y se añade cada uno a la sesión de base de datos utilizando db.add(role).

Confirmación de Cambios:

- Se llama a db.commit() para guardar los cambios en la base de datos.

Refrescar el Estado:

- Se utiliza db.refresh(roles[0]) para actualizar el estado del primer rol en la lista después de la inserción.

Retorno:

- La función devuelve la lista de roles insertados.

```

def seed_roles(db: Session):
    roles = [
        Role(Nombre="Administrador", Descripcion="Usuario con permisos completos para gestionar el sistema."),
        Role(Nombre="Farmacéutico", Descripcion="Responsable de la dispensación y control de medicamentos."),
        Role(Nombre="Cajero", Descripcion="Maneja las transacciones de ventas y cobros en la farmacia."),
        Role(Nombre="Médico", Descripcion="Encargado de realizar consultas y emitir recetas médicas."),
        Role(Nombre="Auxiliar", Descripcion="Brinda apoyo en el inventario y atención al cliente."),
        Role(Nombre="Cliente", Descripcion="Usuario normal")
    ]

    for role in roles:
        db.add(role)

    db.commit()
    db.refresh(roles[0])

    return roles
  
```

Src/init_db.py

Función init_db

- Propósito: Inicializa la base de datos y ejecuta el seeder para insertar roles.

Flujo

- Obtener Sesión de Base de Datos:
- Se crea una sesión de base de datos mediante src.config.db.SessionLocal().

```

# Obtiene una sesión de la base de datos
db = src.config.db.SessionLocal()
  
```

Ejecutar Seeder:

- Se llama a la función seed_roles(db) para insertar los roles predeterminados en la base de datos.
- Se imprime un mensaje de éxito si la inserción se realiza correctamente.

```

# Ejecuta el seeder para los roles
seed_roles(db)
print("Roles inicializados correctamente.")
  
```

Cerrar la Sesión:

- Se asegura de que la sesión de la base de datos se cierre al final de la operación, utilizando finally.

Condición de Ejecución:

- Si el script se ejecuta directamente (no importado como módulo), se llama a init_db() para inicializar la base de datos.

```
if __name__ == "__main__":
    init_db()
```

Src/jwt_config.py

Carga de Configuración

- Se cargan las variables de entorno desde un archivo .env para obtener la clave secreta, el tiempo de expiración y el algoritmo.
- Función solicita_token

Genera un token JWT:

- Payload: Incluye el ID del usuario, el nombre de usuario y la fecha de expiración.
- Expiración: Se calcula agregando horas especificadas a la fecha actual.
- Token: Se codifica utilizando la clave secreta y el algoritmo especificado.
- Función valida_token

Valida un token JWT:

- Decodificación: Intenta decodificar el token y extraer el payload.
- Verificación de Expiración: Comprueba si el token ha expirado comparando la fecha de expiración con la hora actual.
- Errores: Maneja excepciones para tokens expirados o inválidos y lanza HTTPException con un código de estado 401.

Src/main.py

Configuración de la Base de Datos

- Se importa Base para definir todas las tablas y se crea la estructura de la base de datos utilizando Base.metadata.create_all(bind=engine).

Instancia de la Aplicación FastAPI

- Se crea una instancia de la aplicación con un título y una descripción:

```
# Crear la instancia de la aplicación FastAPI
app = FastAPI(
    title="PRESTAMOS S.A. de C.V.",
    description="API para el almacenamiento de información de préstamo de equipo informático",
)
```

Configuración del Middleware CORS

- Se añade middleware CORS para permitir solicitudes de diferentes orígenes:

```
# Configuración del middleware CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Puedes cambiar "*" por ["http://localhost:5500"] si solo quieres permitir ese origen
    allow_credentials=True, # Permitir las credenciales (cookies, tokens, etc.)
    allow_methods=["*"], # Permitir todos los métodos (GET, POST, OPTIONS, etc.)
    allow_headers=["*"], # Permitir todos los encabezados
)
```

Inclusión de Rutas (Routers)

- Se incluyen varios routers en la aplicación para manejar diferentes recursos:

```
# Incluir los routers en la aplicación FastAPI
app.include_router(usuarios_router)
app.include_router(medicamento_router)
app.include_router(receta_router)
app.include_router(dispensacion_router)
app.include_router(lotes_medicamentos_router)
app.include_router(solicitudes_router)
app.include_router(usuarios_roles_router) # Incluir las rutas de usuario_roles
app.include_router(personas_router) # Incluir las rutas de personas
```