FastAPI Ophthalmology Clinic Management System - Complete Guide

This guide will walk you through building a comprehensive Ophthalmology Clinic Management System using FastAPI. We will cover project setup, database schema, API design, authentication, and more.

Table of Contents

1. Project Setup

This section outlines the initial steps to set up your FastAPI project, including virtual environment creation and installing necessary dependencies.

Step 1: Create a New FastAPI Project

First, create a new directory for your project and navigate into it.

Bash

```
mkdir fastapi-ophthalmology-clinic
cd fastapi-ophthalmology-clinic
```

Step 2: Set Up a Virtual Environment

It's good practice to use a virtual environment to manage project dependencies.

Bash

```
python -m venv venv
source venv/bin/activate  # On Windows, use `venv\Scripts\activate`
```

Step 3: Install Required Packages

We'll need FastAPI, Uvicorn (for running the server), Pydantic (for data validation), SQLAlchemy (for ORM), and a database driver (e.g., `psycopg2-binary` for PostgreSQL, `mysql-connector-python` for MySQL, or `sqlite3` which is built-in). For simplicity, we'll start with SQLite.

Bash

```
pip install fastapi uvicorn pydantic sqlalchemy python-jose[cryptography]
passlib[bcrypt]
```

Step 4: Create Main Application File

Create a file named `main.py` at the root of your project. This will be the entry point for your FastAPI application.

Python

```
# main.py
from fastapi import FastAPI


app = FastAPI(title="Ophthalmology Clinic API",
              description="API for managing an Ophthalmology Clinic.",
              version="1.0.0")


@app.get("/")
async def read_root():
    return {"message": "Welcome to the Ophthalmology Clinic API!"}


if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Step 5: Run the Application

You can now run your FastAPI application using Uvicorn.

Bash

```
uvicorn main:app --reload
```

Open your browser and go to `http://127.0.0.1:8000`. You should see the welcome message. You can also access the interactive API documentation at `http://127.0.0.1:8000/docs`.

Project Structure Overview

As your project grows, it's beneficial to organize your code into a logical structure. Here's a suggested layout:

Code

```
fastapi-ophthalmology-clinic/
├── venv/
├── app/
│   ├── api/
│   │   ├── endpoints/          # API endpoint definitions (routers)
│   │   │   ├── users.py
│   │   │   ├── doctors.py
│   │   │   ├── patients.py
│   │   │   ├── appointments.py
│   │   │   ├── consultations.py
│   │   │   ├── optometry.py
│   │   │   ├── pharmacy.py
│   │   │   ├── billing.py
│   │   │   ├── inventory.py
│   │   │   └── __init__.py
│   │   └── __init__.py
│   ├── core/                   # Core application configurations and utilities
│   │   ├── config.py           # Settings and environment variables
│   │   ├── security.py         # Hashing, JWT functions
│   │   └── __init__.py
│   ├── db/                     # Database-related files
│   │   ├── database.py         # SQLAlchemy engine and session
│   │   ├── base.py             # Base for declarative models
│   │   ├── crud.py             # CRUD operations (optional, can be in services)
│   │   └── __init__.py
│   ├── models/                 # SQLAlchemy ORM models
│   │   ├── user.py
│   │   ├── doctor.py
│   │   ├── patient.py
│   │   ├── appointment.py
│   │   ├── consultation.py
│   │   ├── optometry.py
│   │   ├── pharmacy.py
│   │   ├── billing.py
│   │   ├── inventory.py
│   │   └── __init__.py
│   ├── schemas/                # Pydantic models for request/response validation
│   │   ├── user.py
│   │   ├── doctor.py
│   │   ├── patient.py
│   │   ├── appointment.py
│   │   ├── consultation.py
│   │   ├── optometry.py
│   │   ├── pharmacy.py
│   │   ├── billing.py
│   │   ├── inventory.py
│   │   └── __init__.py
│   ├── services/               # Business logic and operations
│   │   ├── user.py
│   │   ├── doctor.py
│   │   ├── patient.py
│   │   ├── appointment.py
│   │   ├── consultation.py
│   │   ├── optometry.py
│   │   ├── pharmacy.py
│   │   ├── billing.py
│   │   ├── inventory.py
│   │   └── __init__.py
│   ├── main.py                 # Main FastAPI application instance
│   └── __init__.py
├── README.md
├── requirements.txt
```

Let's begin structuring our project.

Update `main.py` for structured imports:

Python

```python
# main.py
from fastapi import FastAPI
from app.api.endpoints import users, doctors, patients, appointments, \
    consultations, optometry, pharmacy, billing, inventory


app = FastAPI(title="Ophthalmology Clinic API",
              description="API for managing an Ophthalmology Clinic.",
              version="1.0.0")


# Include routers for different modules
```

```
app.include_router(users.router, prefix="/users", tags=["Users"])
app.include_router(doctors.router, prefix="/doctors", tags=["Doctors"])
app.include_router(patients.router, prefix="/patients", tags=["Patients"])
app.include_router(appointments.router, prefix="/appointments", tags=["Appointments"])
app.include_router(consultations.router, prefix="/consultations",
tags=["Consultations"])
app.include_router(optometry.router, prefix="/optometry", tags=["Optometry"])
app.include_router(pharmacy.router, prefix="/pharmacy", tags=["Pharmacy"])
app.include_router(billing.router, prefix="/billing", tags=["Billing"])
app.include_router(inventory.router, prefix="/inventory", tags=["Inventory"])


@app.get("/", tags=["Root"])
async def read_root():
    return {"message": "Welcome to the Ophthalmology Clinic API!"}
```

Now, create the corresponding empty `__init__.py` files and endpoint files (e.g., `app/api/endpoints/users.py`) for the routers to avoid import errors.

Bash

```bash
mkdir -p app/api/endpoints app/core app/db app/models app/schemas app/services
touch app/__init__.py app/api/__init__.py app/api/endpoints/__init__.py
touch app/api/endpoints/users.py app/api/endpoints/doctors.py
app/api/endpoints/patients.py \
      app/api/endpoints/appointments.py app/api/endpoints/consultations.py \
      app/api/endpoints/optometry.py app/api/endpoints/pharmacy.py \
      app/api/endpoints/billing.py app/api/endpoints/inventory.py
touch app/core/__init__.py app/core/config.py app/core/security.py
touch app/db/__init__.py app/db/database.py app/db/base.py
touch app/models/__init__.py app/models/user.py app/models/doctor.py
app/models/patient.py \
      app/models/appointment.py app/models/consultation.py app/models/optometry.py \
      app/models/pharmacy.py app/models/billing.py app/models/inventory.py
touch app/schemas/__init__.py app/schemas/user.py app/schemas/doctor.py
app/schemas/patient.py \
      app/schemas/appointment.py app/schemas/consultation.py app/schemas/optometry.py
\
      app/schemas/pharmacy.py app/schemas/billing.py app/schemas/inventory.py
touch app/services/__init__.py app/services/user.py app/services/doctor.py
app/services/patient.py \
      app/services/appointment.py app/services/consultation.py
app/services/optometry.py \
      app/services/pharmacy.py app/services/billing.py app/services/inventory.py
```

This initial setup provides a solid foundation for building your FastAPI application.

2. Database Structure & Migrations

In this section, we'll define our database connection, SQLAlchemy base, and how to manage migrations. We'll use SQLite for simplicity during development, but the structure easily adapts to PostgreSQL or MySQL.

Step 1: Database Configuration (`app/core/config.py`)

This file will hold our application settings, including the database URL.

Python

```python
# app/core/config.py
from pydantic_settings import BaseSettings, SettingsConfigDict
import os


class Settings(BaseSettings):
    PROJECT_NAME: str = "Ophthalmology Clinic API"
    API_V1_STR: str = "/api/v1"

    DATABASE_URL: str = os.getenv("DATABASE_URL", "sqlite:///./sql_app.db") # Default
to SQLite

    SECRET_KEY: str = os.getenv("SECRET_KEY", "your-super-secret-key")
    ALGORITHM: str = "HS256"
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30

    model_config = SettingsConfigDict(env_file=".env", extra="ignore")

settings = Settings()
```

Create a `.env` file at the project root to manage environment variables.

Code

```
# .env
DATABASE_URL="sqlite:///./sql_app.db"
SECRET_KEY="your-very-secure-secret-key-that-is-long-and-random"
```

Step 2: Database Connection (app/db/database.py)

This file will set up the SQLAlchemy engine and session.

Python

```python
# app/db/database.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

from app.core.config import settings

# Use connect_args for SQLite to allow multiple threads to access the database
# For other databases like PostgreSQL, you might not need this.
engine = create_engine(
    settings.DATABASE_URL, connect_args={"check_same_thread": False} if "sqlite" in
settings.DATABASE_URL else {}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

# Dependency to get the database session
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Step 3: Base for Models (app/db/base.py)

This file will import all models so that `Base.metadata.create_all` can discover them.

Python

```python
# app/db/base.py
# Import all models here so that Base can discover them
from app.db.database import Base
from app.models.user import User
from app.models.doctor import Doctor
from app.models.patient import Patient
from app.models.appointment import Appointment
from app.models.consultation import Consultation
from app.models.optometry import Optometry
from app.models.pharmacy import Pharmacy
from app.models.billing import Billing
from app.models.inventory import Inventory
```

Step 4: Initial Database Creation

Before defining the models, let's establish how we'll create the database tables.

Add a script to your project root (e.g., `initial_db.py`) to create tables for the first time.

Python

```python
# initial_db.py
from app.db.base import Base
from app.db.database import engine


def create_db_and_tables():
    print("Creating database tables...")
    Base.metadata.create_all(bind=engine)
    print("Database tables created.")


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

Now, when you run `python initial_db.py`, it will create the `sql_app.db` file and all tables defined by your models.

Migrations with Alembic (Optional but Recommended for Production)

For managing database schema changes in a production environment, Alembic is highly recommended. It allows you to track changes to your models and apply them to your database in a controlled way.

Install Alembic:

Bash

```bash
pip install alembic
```

Initialize Alembic:

Bash

```
alembic init alembic
```

This will create an `alembic` directory and `alembic.ini` file.

Configure `alembic.ini`:

Edit `alembic.ini` and set `sqlalchemy.url` to point to your `DATABASE_URL` from `app.core.config`.

Ini

```ini
# alembic.ini
# ...
sqlalchemy.url = sqlite:///./sql_app.db  # Or your PostgreSQL/MySQL URL
# ...
```

Configure `alembic/env.py`:

This file tells Alembic how to connect to your database and discover your models. Modify the `run_migrations_online` function to import your Base from `app.db.base`.

Python

```python
# alembic/env.py
from logging.config import fileConfig

from sqlalchemy import engine_from_config, pool

from alembic import context

# import your Base and engine
from app.db.base import Base
from app.db.database import engine

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This self-contained section covers the autologging feature of Alembic.
if config.config_file_name is not None:
    fileConfig(config.config_file_name)
```

```
# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
target_metadata = Base.metadata # <--- IMPORTANT: Point to your Base.metadata

# other values from the config, defined by the needs of env.py,
# can be acquired as follows:
# my_important_option = config.get_main_option("my_important_option")
# ... etc.

def run_migrations_offline() -> None:
    # ... (unchanged)
    pass

def run_migrations_online() -> None:
    """Run migrations in 'online' mode.

    In this scenario we need to create an Engine
    and associate a connection with the context.

    """
    connectable = engine # <--- IMPORTANT: Use your existing engine

    with connectable.connect() as connection:
        context.configure(
            connection=connection, target_metadata=target_metadata
        )

        with context.begin_transaction():
            context.run_migrations()

if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

Generate Initial Migration:

```
Bash
```

```
alembic revision --autogenerate -m "Initial migration"
```

This will create a new migration file in `alembic/versions`.

Apply Migration:

Bash

```bash
alembic upgrade head
```

This will create your database tables based on your models.

Now you have a robust system for managing your database schema. For any future changes to your models, you'll run `alembic revision --autogenerate -m "Description of change"` and then `alembic upgrade head`.

3. Models & Relationships

Here, we will define the SQLAlchemy ORM models for our clinic management system. Each model will correspond to a table in our database.

Base Model Imports (`app/models/__init__.py`)

This file will ensure all models are imported, allowing `Base.metadata.create_all` and Alembic to discover them.

Python

```python
# app/models/__init__.py
from .user import User
from .doctor import Doctor
from .patient import Patient
from .appointment import Appointment
from .consultation import Consultation
from .optometry import Optometry
from .pharmacy import Pharmacy
from .billing import Billing
from .inventory import Inventory
```

User Model (`app/models/user.py`)

This model will handle clinic staff, including doctors, receptionists, and administrators. It includes fields for authentication.

Python

```python
# app/models/user.py
from sqlalchemy import Column, Integer, String, Boolean, DateTime
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class User(Base):
    __tablename__ = "users"


    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True, nullable=False)
    hashed_password = Column(String, nullable=False)
    first_name = Column(String, nullable=True)
    last_name = Column(String, nullable=True)
    is_active = Column(Boolean, default=True)
    is_superuser = Column(Boolean, default=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())


    # Relationships
    # A user can be associated with a doctor profile (one-to-one)
    doctor_profile = relationship("Doctor", back_populates="user", uselist=False)


    # A user might create/manage appointments, consultations, etc. (one-to-many)
    # This is implicit or can be added explicitly if needed, e.g., created_by_user_id


    def __repr__(self):
        return f"<User {self.email}>"
```

Doctor Model (`app/models/doctor.py`)

This model represents a doctor's specific profile, linked to a User.

Python

```python
# app/models/doctor.py
from sqlalchemy import Column, Integer, String, ForeignKey, Text
from sqlalchemy.orm import relationship
from app.db.database import Base


class Doctor(Base):
    __tablename__ = "doctors"

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey("users.id"), unique=True, nullable=False)
    specialization = Column(String, default="Ophthalmologist")
    license_number = Column(String, unique=True, nullable=True)
    phone_number = Column(String, nullable=True)
    address = Column(String, nullable=True)
    bio = Column(Text, nullable=True)

    # Relationships
    user = relationship("User", back_populates="doctor_profile")
    appointments = relationship("Appointment", back_populates="doctor")
    consultations = relationship("Consultation", back_populates="doctor")

    def __repr__(self):
        return f"<Doctor {self.license_number}>"
```

Patient Model (`app/models/patient.py`)

This model stores patient demographic and contact information.

Python

```python
# app/models/patient.py
from sqlalchemy import Column, Integer, String, Date, Text, DateTime
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base

class Patient(Base):
    __tablename__ = "patients"

    id = Column(Integer, primary_key=True, index=True)
    first_name = Column(String, nullable=False)
    last_name = Column(String, nullable=False)
    date_of_birth = Column(Date, nullable=False)
    gender = Column(String, nullable=True) # e.g., "Male", "Female", "Other"
    phone_number = Column(String, nullable=True)
    email = Column(String, unique=True, index=True, nullable=True)
    address = Column(String, nullable=True)
    medical_history = Column(Text, nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())
    # Relationships
    appointments = relationship("Appointment", back_populates="patient")
    consultations = relationship("Consultation", back_populates="patient")
    optometry_records = relationship("Optometry", back_populates="patient")
    billings = relationship("Billing", back_populates="patient")
    prescriptions = relationship("Pharmacy", back_populates="patient") # linking
pharmacy to patient via prescriptions

    def __repr__(self):
        return f"<Patient {self.first_name} {self.last_name}>"
```

Appointment Model (`app/models/appointment.py`)

This model manages patient appointments with doctors.

Python

```
# app/models/appointment.py
from sqlalchemy import Column, Integer, String, DateTime, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class Appointment(Base):
    __tablename__ = "appointments"

    id = Column(Integer, primary_key=True, index=True)
    patient_id = Column(Integer, ForeignKey("patients.id"), nullable=False)
    doctor_id = Column(Integer, ForeignKey("doctors.id"), nullable=False)
    appointment_time = Column(DateTime(timezone=True), nullable=False)
    status = Column(String, default="Scheduled") # e.g., "Scheduled", "Completed",
"Cancelled", "No-Show"
    reason_for_visit = Column(Text, nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

    # Relationships
    patient = relationship("Patient", back_populates="appointments")
    doctor = relationship("Doctor", back_populates="appointments")

    def __repr__(self):
        return f"<Appointment {self.id} for Patient {self.patient_id} with Doctor
{self.doctor_id}>"
```

Consultation Model (`app/models/consultation.py`)

This model stores detailed records of a doctor-patient consultation.

Python

```python
# app/models/consultation.py
from sqlalchemy import Column, Integer, String, Text, DateTime, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class Consultation(Base):
    __tablename__ = "consultations"


    id = Column(Integer, primary_key=True, index=True)
    patient_id = Column(Integer, ForeignKey("patients.id"), nullable=False)
    doctor_id = Column(Integer, ForeignKey("doctors.id"), nullable=False)
    consultation_date = Column(DateTime(timezone=True), server_default=func.now())
    symptoms = Column(Text, nullable=True)
    diagnosis = Column(Text, nullable=True)
    treatment_plan = Column(Text, nullable=True)
    notes = Column(Text, nullable=True)
    follow_up_date = Column(DateTime(timezone=True), nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())



    # Relationships
    patient = relationship("Patient", back_populates="consultations")
    doctor = relationship("Doctor", back_populates="consultations")
    optometry_record = relationship("Optometry", back_populates="consultation",
uselist=False)
    prescription = relationship("Pharmacy", back_populates="consultation",
uselist=False)

    def __repr__(self):
        return f"<Consultation {self.id} on {self.consultation_date.date()}>"
```

Optometry Model (`app/models/optometry.py`)

This model will store specific optometry examination results.

Python

```python
# app/models/optometry.py
from sqlalchemy import Column, Integer, String, Text, DateTime, ForeignKey, Float
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class Optometry(Base):
    __tablename__ = "optometry_records"


    id = Column(Integer, primary_key=True, index=True)
    patient_id = Column(Integer, ForeignKey("patients.id"), nullable=False)
    consultation_id = Column(Integer, ForeignKey("consultations.id"), unique=True,
nullable=True) # Link to a specific consultation
    exam_date = Column(DateTime(timezone=True), server_default=func.now())
    visual_acuity_od = Column(String, nullable=True) # OD: Oculus Dexter (Right Eye)
    visual_acuity_os = Column(String, nullable=True) # OS: Oculus Sinister (Left Eye)
    refraction_od = Column(String, nullable=True)
    refraction_os = Column(String, nullable=True)
    intraocular_pressure_od = Column(Float, nullable=True)
    intraocular_pressure_os = Column(Float, nullable=True)
    fundus_exam_notes = Column(Text, nullable=True)
    slit_lamp_exam_notes = Column(Text, nullable=True)
    field_of_vision_test_results = Column(Text, nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())



    # Relationships
    patient = relationship("Patient", back_populates="optometry_records")
    consultation = relationship("Consultation", back_populates="optometry_record")


    def __repr__(self):
        return f"<Optometry Record {self.id} for Patient {self.patient_id}>"
```

Pharmacy Model (app/models/pharmacy.py)

This model manages prescriptions issued during consultations.

Python

```python
# app/models/pharmacy.py
from sqlalchemy import Column, Integer, String, Text, DateTime, ForeignKey, Float
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class Pharmacy(Base):
    __tablename__ = "prescriptions" # Renamed to 'prescriptions' for clarity


    id = Column(Integer, primary_key=True, index=True)
    patient_id = Column(Integer, ForeignKey("patients.id"), nullable=False)
    consultation_id = Column(Integer, ForeignKey("consultations.id"), unique=True,
nullable=False)
    medication_name = Column(String, nullable=False)
    dosage = Column(String, nullable=True)
    frequency = Column(String, nullable=True) # e.g., "Once Daily", "Twice A Day"
    duration = Column(String, nullable=True) # e.g., "7 days", "Until finished"
    quantity = Column(Integer, nullable=True)
    instructions = Column(Text, nullable=True)
    prescribed_date = Column(DateTime(timezone=True), server_default=func.now())
    pharmacist_notes = Column(Text, nullable=True)
    dispensed_date = Column(DateTime(timezone=True), nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())


    # Relationships
    patient = relationship("Patient", back_populates="prescriptions")
    consultation = relationship("Consultation", back_populates="prescription")
    # Inventory can be linked here to track dispensed items
    # e.g., prescribed_items = relationship("InventoryItem",
secondary=prescription_item_association_table)


    def __repr__(self):
        return f"<Prescription {self.id} for {self.medication_name}>"
```

Billing Model (app/models/billing.py)

This model handles invoicing and payment records.

Python

```
# app/models/billing.py
from sqlalchemy import Column, Integer, String, Text, DateTime, ForeignKey, Float
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class Billing(Base):
    __tablename__ = "billings"


    id = Column(Integer, primary_key=True, index=True)
    patient_id = Column(Integer, ForeignKey("patients.id"), nullable=False)
    consultation_id = Column(Integer, ForeignKey("consultations.id"), nullable=True) #
Link to a specific consultation if applicable
    invoice_date = Column(DateTime(timezone=True), server_default=func.now())
    total_amount = Column(Float, nullable=False)
    amount_paid = Column(Float, default=0.0)
    payment_status = Column(String, default="Pending") # e.g., "Pending", "Paid",
"Partially Paid", "Cancelled"
    payment_method = Column(String, nullable=True) # e.g., "Cash", "Card", "Insurance"
    due_date = Column(DateTime(timezone=True), nullable=True)
    notes = Column(Text, nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())


    # Relationships
    patient = relationship("Patient", back_populates="billings")
    # A billing record might be associated with a specific consultation or multiple
services/items.
    # For simplicity, we'll link to consultation and assume services/items are handled
within notes or a separate line_items table.


    def __repr__(self):
        return f"<Billing {self.id} for Patient {self.patient_id}>"
```

Inventory Model (`app/models/inventory.py`)

This model tracks medical supplies and medications.

Python

```python
# app/models/inventory.py
from sqlalchemy import Column, Integer, String, Text, DateTime, Float
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from app.db.database import Base


class Inventory(Base):
    __tablename__ = "inventory"


    id = Column(Integer, primary_key=True, index=True)
    item_name = Column(String, unique=True, index=True, nullable=False)
    item_type = Column(String, nullable=True) # e.g., "Medication", "Lens",
"Equipment"


    description = Column(Text, nullable=True)
    quantity_on_hand = Column(Integer, default=0)
    reorder_level = Column(Integer, default=10)
    unit_price = Column(Float, nullable=True)
    supplier = Column(String, nullable=True)
    expiry_date = Column(DateTime(timezone=True), nullable=True)
    location = Column(String, nullable=True) # e.g., "Storage Room A", "Pharmacy Shelf
3"
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())


    # Relationships (if linking inventory movements or specific item dispensing)
    # E.g., usage_records = relationship("InventoryUsage", back_populates="item")


    def __repr__(self):
        return f"<Inventory Item: {self.item_name}>"
```

4. Authentication & Authorization

- **Authentication:** Verifying who the user is.
    - **Methods:** Token-based (JWT is common with FastAPI), OAuth2.
    - **Libraries:** `python-jose` for JWTs, `passlib` for password hashing.
    - **Flow:** User sends credentials, receives a JWT, includes JWT in subsequent requests.
- **Authorization:** Determining what an authenticated user can do.
    - **Methods:** Role-based access control (RBAC), permission-based access control.
    - **Implementation:** Decorators or dependencies in FastAPI to check user roles/permissions against required permissions for an endpoint.

5. Controllers & API Endpoints

- **Controllers (or Routers in FastAPI):** Groups related endpoints.
    - **Purpose:** Organize your API, define request/response models.
- **API Endpoints:** Specific URLs that perform actions.
    - **Examples:**
        - `POST /users/register` (Register a new user)
        - `POST /users/login` (Authenticate a user)
        - `GET /patients/` (Get all patients)
        - `POST /patients/` (Create a new patient record)
        - `GET /patients/{patient_id}` (Get a specific patient)
        - `PUT /patients/{patient_id}` (Update a patient)
        - `DELETE /patients/{patient_id}` (Delete a patient)
        - Similar endpoints for appointments, medical records, doctors, etc.

6. Routes & API Structure

- **FastAPI's `APIRouter`:** Essential for modularizing your API.
    - Create separate routers for `users`, `patients`, `appointments`, etc.
    - Include these routers in your main FastAPI application.
- **Directory Structure:**

Code

```
fastapi-ophthalmology-clinic/
├── venv/
├── app/
│   ├── api/
│   │   ├── endpoints/          # API endpoint definitions (routers)
│   │   │   ├── users.py
│   │   │   ├── doctors.py
│   │   │   ├── patients.py
│   │   │   ├── appointments.py
│   │   │   ├── consultations.py
│   │   │   ├── optometry.py
│   │   │   ├── pharmacy.py
│   │   │   ├── billing.py
│   │   │   ├── inventory.py
│   │   │   └── __init__.py
│   │   └── __init__.py
```

7. Patient Flow Implementation

This is where you define the business logic for managing patient data and clinic operations.

- **Patient Registration:** Create patient records, potentially link to user accounts.
- **Appointment Scheduling:** Create, view, update, and cancel appointments.
- **Medical Records:** Store and retrieve patient history, diagnoses, prescriptions.
- **Billing (Optional):** Integrate with a billing system or manage basic invoices.
- **Doctor Assignment:** Assign doctors to patients or appointments.

8. Testing with Postman

- **Verify Endpoints:** Send requests to your API endpoints to ensure they behave as expected.
- **Authentication Testing:** Test login, use the obtained JWT for authenticated requests.
- **CRUD Operations:** Test creating, reading, updating, and deleting records.
- **Error Handling:** Test invalid inputs, unauthorized access, and other error scenarios.
- **Collections:** Organize your requests into Postman collections for easier management.

9. Deployment Considerations

- **Web Server:** Use an ASGI server like Uvicorn to serve your FastAPI application.
- **Process Manager:** Gunicorn (often paired with Uvicorn workers) for managing multiple processes.
- **Environment Variables:** Store sensitive information (database credentials, JWT secret) in environment variables.
- **Containerization (Docker):** Package your application and its dependencies into a Docker image for consistent deployment.
- **Orchestration (Kubernetes):** For larger deployments, manage containers with Kubernetes.
- **Cloud Providers:** Deploy on platforms like AWS, Google Cloud, Azure, Heroku, or Render.
- **Database Setup:** Ensure your PostgreSQL/MySQL database is properly provisioned and accessible.

- **CI/CD:** Implement continuous integration and continuous deployment pipelines for automated testing and deployment.