Sistema de Hotelaria — Projeto 3 camadas (FastAPI + SQLite)

Documento completo com estrutura de projeto, todos os arquivos de código, e explicação passo-a-passo **linha a linha** para você copiar.

Visão geral

- Linguagem: Python 3.11+
- Backend: FastAPI
- ORM: SQLAlchemy
- Banco de dados: SQLite (arquivo `hotel.db`)
- Arquitetura: 3 camadas
- **Apresentação (controllers / rotas)**: `app/routes.py` + `frontend/index.html`
- **Serviço / Negócio**: `app/services.py`
- **Acesso a dados (repositório / models)**: `app/repositories.py`, `app/models.py`, `app/database.py`

- Mínimo 6 tabelas (criadas como modelos SQLAlchemy):

- `Hotel`
- `RoomType`
- `Room`
- `Guest`
- `Reservation`
- `Payment`
- `Staff` (opcional, incluído para completar)

Estrutura do projeto (arquivos)

```
hotel-system/
├─ app/
│  ├─ __init__.py
│  ├─ main.py
│  ├─ database.py
│  ├─ models.py
│  ├─ schemas.py
│  ├─ repositories.py
│  ├─ services.py
│  └─ routes.py
├─ frontend/
│  └─ index.html
```

```
├─ requirements.txt
└─ README.md
```

---

requirements.txt

```
fastapi
uvicorn[standard]
SQLAlchemy
pydantic
alembic
databases
python-dotenv
```

Pode instalar com: `pip install -r requirements.txt`.

---

app/database.py

```python
# app/database.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base
import os

DB_PATH = os.getenv('HOTEL_DB', 'sqlite:///./hotel.db')

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

def init_db():
    # cria as tabelas no banco (se não existirem)
    Base.metadata.create_all(bind=engine)
```

Explicação (linha a linha) - `create_engine(...)`: configura o SQLAlchemy para usar SQLite local. `check_same_thread=False` é necessário para SQLite quando usamos sessions em FastAPI. - `SessionLocal`: fabrica sessões (DB sessions) que serão injetadas no repositório. - `Base`: classe base para modelos declarativos. - `init_db()`: função utilitária para criar as tabelas no primeiro run.

---

app/models.py

```python
# app/models.py
from sqlalchemy import Column, Integer, String, DateTime, ForeignKey, Float,
Boolean, Text
```

```python
from sqlalchemy.orm import relationship
from .database import Base
import datetime

class Hotel(Base):
    __tablename__ = 'hotels'
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    address = Column(String, nullable=True)
    city = Column(String, nullable=True)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)

    rooms = relationship('Room', back_populates='hotel')

class RoomType(Base):
    __tablename__ = 'room_types'
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    description = Column(Text, nullable=True)
    price = Column(Float, nullable=False)

    rooms = relationship('Room', back_populates='room_type')

class Room(Base):
    __tablename__ = 'rooms'
    id = Column(Integer, primary_key=True, index=True)
    number = Column(String, nullable=False)
    floor = Column(Integer, nullable=True)
    hotel_id = Column(Integer, ForeignKey('hotels.id'))
    room_type_id = Column(Integer, ForeignKey('room_types.id'))
    is_available = Column(Boolean, default=True)

    hotel = relationship('Hotel', back_populates='rooms')
    room_type = relationship('RoomType', back_populates='rooms')
    reservations = relationship('Reservation', back_populates='room')

class Guest(Base):
    __tablename__ = 'guests'
    id = Column(Integer, primary_key=True, index=True)
    first_name = Column(String, nullable=False)
    last_name = Column(String, nullable=False)
    email = Column(String, nullable=True)
    phone = Column(String, nullable=True)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)

    reservations = relationship('Reservation', back_populates='guest')

class Reservation(Base):
    __tablename__ = 'reservations'
    id = Column(Integer, primary_key=True, index=True)
    guest_id = Column(Integer, ForeignKey('guests.id'))
```

```
    room_id = Column(Integer, ForeignKey('rooms.id'))
    check_in = Column(DateTime, nullable=False)
    check_out = Column(DateTime, nullable=False)
    total_amount = Column(Float, nullable=False)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)

    guest = relationship('Guest', back_populates='reservations')
    room = relationship('Room', back_populates='reservations')
    payments = relationship('Payment', back_populates='reservation')

class Payment(Base):
    __tablename__ = 'payments'
    id = Column(Integer, primary_key=True, index=True)
    reservation_id = Column(Integer, ForeignKey('reservations.id'))
    amount = Column(Float, nullable=False)
    paid_at = Column(DateTime, default=datetime.datetime.utcnow)
    method = Column(String, nullable=True)

    reservation = relationship('Reservation', back_populates='payments')

class Staff(Base):
    __tablename__ = 'staff'
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    role = Column(String, nullable=True)
    email = Column(String, nullable=True)
```

Explicação (resumo) - Definimos 7 tabelas. Cada `relationship` ajuda o SQLAlchemy a navegar entre objetos. - Campos `created_at` guardam quando foi criado o registro.

---

app/schemas.py (Pydantic)

```
# app/schemas.py
from pydantic import BaseModel
from typing import Optional
import datetime

# Schemas simples para entrada/saída
class HotelCreate(BaseModel):
    name: str
    address: Optional[str] = None
    city: Optional[str] = None

class HotelRead(HotelCreate):
    id: int
    created_at: datetime.datetime
    class Config:
        orm_mode = True
```

```python
class RoomTypeCreate(BaseModel):
    name: str
    description: Optional[str] = None
    price: float

class RoomTypeRead(RoomTypeCreate):
    id: int
    class Config:
        orm_mode = True

class RoomCreate(BaseModel):
    number: str
    floor: Optional[int] = None
    hotel_id: int
    room_type_id: int

class RoomRead(RoomCreate):
    id: int
    is_available: bool
    class Config:
        orm_mode = True

class GuestCreate(BaseModel):
    first_name: str
    last_name: str
    email: Optional[str] = None
    phone: Optional[str] = None

class GuestRead(GuestCreate):
    id: int
    created_at: datetime.datetime
    class Config:
        orm_mode = True

class ReservationCreate(BaseModel):
    guest_id: int
    room_id: int
    check_in: datetime.datetime
    check_out: datetime.datetime
    total_amount: float

class ReservationRead(ReservationCreate):
    id: int
    created_at: datetime.datetime
    class Config:
        orm_mode = True

class PaymentCreate(BaseModel):
    reservation_id: int
    amount: float
```

```
    method: Optional[str] = None

class PaymentRead(PaymentCreate):
    id: int
    paid_at: datetime.datetime
    class Config:
        orm_mode = True
```

---

app/repositories.py (Acesso a dados)

```
# app/repositories.py
from sqlalchemy.orm import Session
from . import models

# Exemplos de funções CRUD simples — o Service chamará essas funções

def create_hotel(db: Session, name: str, address: str | None, city: str |
None) -> models.Hotel:
    h = models.Hotel(name=name, address=address, city=city)
    db.add(h)
    db.commit()
    db.refresh(h)
    return h

def get_hotels(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Hotel).offset(skip).limit(limit).all()

# Guests
def create_guest(db: Session, guest: models.Guest) -> models.Guest:
    db.add(guest)
    db.commit()
    db.refresh(guest)
    return guest

def get_guest(db: Session, guest_id: int):
    return db.query(models.Guest).filter(models.Guest.id == guest_id).first()

# Rooms
def create_room(db: Session, room: models.Room) -> models.Room:
    db.add(room)
    db.commit()
    db.refresh(room)
    return room

def get_available_rooms(db: Session):
    return db.query(models.Room).filter(models.Room.is_available ==
True).all()
```

```
# Reservations
def create_reservation(db: Session, reservation: models.Reservation) ->
models.Reservation:
    db.add(reservation)
    db.commit()
    db.refresh(reservation)
    return reservation

# Payments
def create_payment(db: Session, payment: models.Payment) -> models.Payment:
    db.add(payment)
    db.commit()
    db.refresh(payment)
    return payment
```

Explicação - Essas funções encapsulam operações com `db` (Session). Mantemos o `commit()` e `refresh()` para garantir que retornamos a entidade com seu `id` preenchido.

---

app/services.py (Lógica de negócio)

```
# app/services.py
from sqlalchemy.orm import Session
from . import repositories, models
import datetime

# Serviço para criar uma reserva: valida disponibilidade e marca room como
indisponível

def make_reservation(db: Session, guest_id: int, room_id: int, check_in:
datetime.datetime, check_out: datetime.datetime, total_amount: float):
    # validações: check_in < check_out
    if check_in >= check_out:
        raise ValueError('check_in deve ser anterior a check_out')

    room = db.query(models.Room).filter(models.Room.id == room_id).first()
    if not room:
        raise ValueError('Quarto não encontrado')
    if not room.is_available:
        raise ValueError('Quarto não disponível')

    reservation = models.Reservation(
        guest_id=guest_id,
        room_id=room_id,
        check_in=check_in,
        check_out=check_out,
        total_amount=total_amount,
        created_at=datetime.datetime.utcnow()
    )
```

```
    # cria reserva
    reservation = repositories.create_reservation(db, reservation)

    # marca quarto como indisponível
    room.is_available = False
    db.add(room)
    db.commit()
    db.refresh(room)

    return reservation
```

Explicação - Serviço realiza validações e chama o repositório para persistir. Além disso altera estado do `room`.

---

app/routes.py (Controllers / API)

```
# app/routes.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from . import schemas, database, repositories, services, models
from typing import List
import datetime

router = APIRouter()

# dependency para obter DB session

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

@router.post('/hotels', response_model=schemas.HotelRead)
def create_hotel(h: schemas.HotelCreate, db: Session = Depends(get_db)):
    return repositories.create_hotel(db, h.name, h.address, h.city)

@router.get('/hotels', response_model=List[schemas.HotelRead])
def list_hotels(db: Session = Depends(get_db)):
    return repositories.get_hotels(db)

@router.post('/guests', response_model=schemas.GuestRead)
def create_guest(g: schemas.GuestCreate, db: Session = Depends(get_db)):
    guest = models.Guest(**g.dict())
    return repositories.create_guest(db, guest)

@router.get('/rooms/available', response_model=List[schemas.RoomRead])
```

```
def available_rooms(db: Session = Depends(get_db)):
    return repositories.get_available_rooms(db)

@router.post('/reservations', response_model=schemas.ReservationRead)
def create_reservation(r: schemas.ReservationCreate, db: Session =
Depends(get_db)):
    try:
        res = services.make_reservation(db, r.guest_id, r.room_id,
r.check_in, r.check_out, r.total_amount)
        return res
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))

@router.post('/payments', response_model=schemas.PaymentRead)
def pay(p: schemas.PaymentCreate, db: Session = Depends(get_db)):
    # verifica reserva
    res = db.query(models.Reservation).filter(models.Reservation.id ==
p.reservation_id).first()
    if not res:
        raise HTTPException(status_code=404, detail='Reserva não encontrada')
    payment = models.Payment(**p.dict())
    return repositories.create_payment(db, payment)
```

Explicação - Roteamos endpoints REST básicos para criar hotéis, hóspedes, reservas e pagamentos. - `get_db` provê uma Session por-request.

---

app/main.py

```
# app/main.py
from fastapi import FastAPI
from .routes import router
from .database import init_db

app = FastAPI(title='Hotel System')
app.include_router(router)

@app.on_event('startup')
def on_startup():
    init_db()
```

Explicação - Cria a aplicação FastAPI e chama `init_db()` ao iniciar para garantir criação das tabelas.

---

frontend/index.html

```
<!doctype html>
<html>
```

```
<head>
  <meta charset="utf-8">
  <title>Hotel System - Frontend</title>
</head>
<body>
  <h1>Sistema de Hotelaria (Frontend simples)</h1>
  <form id="hotelForm">
    <input name="name" placeholder="Nome do hotel" required />
    <input name="city" placeholder="Cidade" />
    <button type="submit">Criar hotel</button>
  </form>
  <ul id="hotels"></ul>

  <script>
  const apiBase = '/api'; // quando for deploy certifique que aponto para o
backend

  async function listHotels(){
    const res = await fetch(apiBase + '/hotels')
    const data = await res.json()
    const ul = document.getElementById('hotels')
    ul.innerHTML = ''
    data.forEach(h=>{
      const li = document.createElement('li')
      li.textContent = `${h.id} - ${h.name} (${h.city || 'sem cidade'})`
      ul.appendChild(li)
    })
  }

  document.getElementById('hotelForm').addEventListener('submit', async
(e)=>{
    e.preventDefault()
    const fd = new FormData(e.target)
    const body = {name: fd.get('name'), city: fd.get('city')}
    await fetch(apiBase + '/hotels', {
      method: 'POST', headers: {'Content-Type':'application/json'}, body:
JSON.stringify(body)
    })
    await listHotels()
  })

  listHotels()
  </script>
</body>
</html>
```

Observação: Para servir esse `index.html`, você pode usar um servidor estático simples (ex: `uvicorn` serve só o backend) — ou usar `app.mount('/frontend', StaticFiles(...))` no FastAPI.

Como rodar (passo a passo comandos)

1. Crie e ative virtualenv (opcional):

```
python -m venv venv
source venv/bin/activate    # linux / mac
venv\Scripts\activate     # windows
```

1. Instale dependências:

```
pip install -r requirements.txt
```

2. Execute a aplicação:

```
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

3. Abra o navegador em `http://127.0.0.1:8000/docs` para ver a documentação automática (Swagger UI) e testar rotas.

4. Para testar o frontend localmente: abra `frontend/index.html` diretamente (ou sirva-o com `python -m http.server 3000` e atualize `apiBase` para `http://127.0.0.1:8000`).

Exemplos `curl`

Criar hotel:

```
curl -X POST "http://127.0.0.1:8000/hotels" -H "Content-Type: application/
json" -d '{"name":"Hotel Exemplo","city":"Recife"}'
```

Criar hóspede:

```
curl -X POST "http://127.0.0.1:8000/guests" -H "Content-Type: application/
json" -d '{"first_name":"Joao","last_name":"Silva","email":"j@x.com"}'
```

Listar quartos disponíveis:

```
curl http://127.0.0.1:8000/rooms/available
```

Fazer reserva (data no formato ISO):

```
curl -X POST "http://127.0.0.1:8000/reservations" -H "Content-Type:
application/json" -d '{"guest_id":1,"room_id":
1,"check_in":"2025-12-01T14:00:00","check_out":"2025-12-05T12:00:00","total_amount":
400.0}'
```

Boas práticas e próximos passos

- Adicionar autenticação (JWT) para rotas administrativas.
- Criar testes unitários (pytest).
- Usar Alembic para migrações.
- Tratar concorrência/lock quando várias pessoas tentam reservar o mesmo quarto ao mesmo tempo (transações ou bloqueio otimista/pessimista).
- Melhorar frontend: React ou Vue para uma interface completa.

Observações finais

- Este projeto é um ponto de partida funcional para um sistema de hotelaria em arquitetura 3 camadas usando FastAPI + SQLite.
- O código aqui está intencionalmente simples e didático: você pode copiar os arquivos para a sua máquina e executar os passos acima.

Se quiser, eu posso: - Gerar os arquivos prontos para download (zip) ou - Subir para um repositório no GitHub (se você me fornecer o repo ou permissões) ou - Converter o frontend para React.

Fim do documento.