Final-algoritmos3/

controllers/

database/

models/

repositories/

doc/

app.py

managers.py

models.py

Archivos ya desarrollados:

controllers:

auth\_controller.py:

from flask import Blueprint, request

from repositories.user\_repo import UserRepo

bp = Blueprint("auth", \_\_name\_\_, url\_prefix="/auth")

repo = UserRepo()

@bp.post("/register")

def register():

    d = request.get\_json(force=True, silent=True) or request.form

    uid = repo.create(d["username"], d["password"], d["email"], d.get("name"))

    return {"id": uid}, 201

categori\_controller.py:

from flask import Blueprint, request

from repositories.category\_repo import CategoryRepo

bp = Blueprint("categories", \_\_name\_\_, url\_prefix="/categories")

repo = CategoryRepo()

@bp.post("")

def create\_category():

    d = request.get\_json(force=True, silent=True) or request.form

    cid = repo.create(d["name"], d.get("description"))

    return {"id": cid}, 201

@bp.get("")

def list\_categories():

    return repo.list()

@bp.patch("/<int:category\_id>")

def update\_category(category\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.update(category\_id, d["name"], d.get("description"))

    return {"ok": True}

@bp.delete("/<int:category\_id>")

def delete\_category(category\_id:int):

    repo.delete(category\_id)

    return {"ok": True}

client\_controller.py:

from flask import Blueprint, request

from repositories.client\_repo import ClientRepo

bp = Blueprint("clients", \_\_name\_\_, url\_prefix="/clients")

repo = ClientRepo()

@bp.post("")

def create\_client():

    d = request.get\_json(force=True, silent=True) or request.form

    cid = repo.create(d["name"], d["email"], d.get("phone"), d.get("address"))

    return {"id": cid}, 201

@bp.get("")

def list\_clients():

    return repo.list()

@bp.get("/<int:client\_id>")

def get\_client(client\_id: int):

    return repo.get(client\_id) or ({"error": "not found"}, 404)

@bp.patch("/<int:client\_id>")

def update\_client(client\_id: int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.update(client\_id, \*\*d)

    return {"ok": True}

@bp.delete("/<int:client\_id>")

def delete\_client(client\_id: int):

    repo.delete(client\_id)

    return {"ok": True}

employee\_controller:

from flask import Blueprint, request

from repositories.employee\_repo import EmployeeRepo

bp = Blueprint("employees", \_\_name\_\_, url\_prefix="/employees")

repo = EmployeeRepo()

@bp.post("")

def create\_employee():

    d = request.get\_json(force=True, silent=True) or request.form

    eid = repo.create(d["name"], d["email"], d["employee\_type"], d.get("employee\_category\_id"), d.get("user\_id"))

    return {"id": eid}, 201

@bp.get("")

def list\_employees():

    return repo.list()

@bp.get("/<int:employee\_id>")

def get\_employee(employee\_id:int):

    return repo.get(employee\_id) or ({"error":"not found"}, 404)

@bp.patch("/<int:employee\_id>")

def update\_employee(employee\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.update(employee\_id, \*\*d)

    return {"ok": True}

@bp.delete("/<int:employee\_id>")

def delete\_employee(employee\_id:int):

    repo.delete(employee\_id)

    return {"ok": True}

equipment\_category\_controller.py:  
  
from flask import Blueprint, request

from repositories.equipment\_category\_repo import EquipmentCategoryRepo

bp = Blueprint("equipment\_categories", \_\_name\_\_, url\_prefix="/equipment-categories")

repo = EquipmentCategoryRepo()

@bp.post("")

def create\_equipment\_category():

    d = request.get\_json(force=True, silent=True) or request.form

    eid = repo.create(d["name"], d.get("description"))

    return {"id": eid}, 201

@bp.get("")

def list\_equipment\_categories():

    return repo.list()

@bp.patch("/<int:ec\_id>")

def update\_equipment\_category(ec\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.update(ec\_id, d["name"], d.get("description"))

    return {"ok": True}

@bp.delete("/<int:ec\_id>")

def delete\_equipment\_category(ec\_id:int):

    repo.delete(ec\_id)

    return {"ok": True}

equipment\_controller.py:

from flask import Blueprint, request

from repositories.equipment\_repo import EquipmentRepo

bp = Blueprint("equipment", \_\_name\_\_, url\_prefix="/equipment")

repo = EquipmentRepo()

@bp.post("")

def create\_equipment():

    d = request.get\_json(force=True, silent=True) or request.form

    eid = repo.create(d["name"], d["serial"], d.get("equipment\_category\_id"))

    return {"id": eid}, 201

@bp.get("")

def list\_equipment():

    return repo.list()

@bp.patch("/<int:equipment\_id>")

def update\_equipment(equipment\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.update(equipment\_id, \*\*d)

    return {"ok": True}

@bp.delete("/<int:equipment\_id>")

def delete\_equipment(equipment\_id:int):

    repo.delete(equipment\_id)

    return {"ok": True}

incident\_controller.py:

from flask import Blueprint, request

from repositories.incident\_repo import IncidentRepo

bp = Blueprint("incidents", \_\_name\_\_, url\_prefix="/incidents")

repo = IncidentRepo()

@bp.post("")

def create\_incident():

    d = request.get\_json(force=True, silent=True) or request.form

    iid = repo.create(int(d["ticket\_id"]), d["description"])

    return {"id": iid}, 201

@bp.get("/by-ticket/<int:ticket\_id>")

def list\_incidents(ticket\_id:int):

    return repo.list\_by\_ticket(ticket\_id)

messege\_controller.py:

from flask import Blueprint, request

from repositories.message\_repo import MessageRepo

bp = Blueprint("messages", \_\_name\_\_, url\_prefix="/messages")

repo = MessageRepo()

@bp.post("")

def create\_message():

    d = request.get\_json(force=True, silent=True) or request.form

    mid = repo.create(int(d["ticket\_id"]), int(d["sender\_employee\_id"]), d["content"])

    return {"id": mid}, 201

@bp.get("/by-ticket/<int:ticket\_id>")

def list\_messages(ticket\_id:int):

    return repo.list\_by\_ticket(ticket\_id)

service\_controller.py:

from flask import Blueprint, request

from repositories.message\_repo import MessageRepo

bp = Blueprint("messages", \_\_name\_\_, url\_prefix="/messages")

repo = MessageRepo()

@bp.post("")

def create\_message():

    d = request.get\_json(force=True, silent=True) or request.form

    mid = repo.create(int(d["ticket\_id"]), int(d["sender\_employee\_id"]), d["content"])

    return {"id": mid}, 201

@bp.get("/by-ticket/<int:ticket\_id>")

def list\_messages(ticket\_id:int):

    return repo.list\_by\_ticket(ticket\_id)

service\_controller.py:

from flask import Blueprint, request

from repositories.service\_repo import ServiceRepo

bp = Blueprint("services", \_\_name\_\_, url\_prefix="/services")

repo = ServiceRepo()

@bp.post("")

def create\_service():

    d = request.get\_json(force=True, silent=True) or request.form

    sid = repo.create(d["name"], d.get("description"), d.get("category\_id"))

    return {"id": sid}, 201

@bp.get("")

def list\_services():

    return repo.list()

@bp.patch("/<int:service\_id>")

def update\_service(service\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.update(service\_id, \*\*d)

    return {"ok": True}

@bp.delete("/<int:service\_id>")

def delete\_service(service\_id:int):

    repo.delete(service\_id)

    return {"ok": True}

team\_controller.py:

from flask import Blueprint, request

from repositories.team\_repo import TeamRepo

bp = Blueprint("teams", \_\_name\_\_, url\_prefix="/teams")

repo = TeamRepo()

@bp.post("")

def create\_team():

    d = request.get\_json(force=True, silent=True) or request.form

    tid = repo.create(d["name"], d.get("description"))

    return {"id": tid}, 201

@bp.post("/<int:team\_id>/members")

def add\_member(team\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.add\_member(team\_id, int(d["employee\_id"]))

    return {"ok": True}

@bp.get("")

def list\_teams():

    return repo.list()

ticket\_controller.py:

from flask import Blueprint, request

from repositories.ticket\_repo import TicketRepo

bp = Blueprint("tickets", \_\_name\_\_, url\_prefix="/tickets")

repo = TicketRepo()

@bp.post("")

def open\_ticket():

    d = request.get\_json(force=True, silent=True) or request.form

    tid = repo.create(

        client\_id=int(d["client\_id"]),

        reporter\_employee\_id=d.get("reporter\_employee\_id"),

        team\_id=d.get("team\_id"),

        state="OPEN",

    )

    return {"id": tid}, 201

@bp.post("/<int:ticket\_id>/employees")

def link\_employee(ticket\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.add\_employee(ticket\_id, int(d["employee\_id"]))

    return {"ok": True}

@bp.post("/<int:ticket\_id>/services")

def link\_service(ticket\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.add\_service(ticket\_id, int(d["service\_id"]))

    return {"ok": True}

@bp.patch("/<int:ticket\_id>/state")

def set\_state(ticket\_id:int):

    d = request.get\_json(force=True, silent=True) or request.form

    repo.set\_state(ticket\_id, d["state"])

    return {"ok": True}

@bp.get("")

def list\_tickets():

    return repo.list()

@bp.get("/<int:ticket\_id>")

def get\_ticket(ticket\_id:int):

    return repo.get(ticket\_id) or ({"error":"not found"}, 404)

work\_controller.py:

from flask import Blueprint, request

from repositories.work\_repo import WorkRepo

bp = Blueprint("work", \_\_name\_\_, url\_prefix="/work")

repo = WorkRepo()

@bp.post("")

def create\_work():

    d = request.get\_json(force=True, silent=True) or request.form

    wid = repo.create(

        int(d["technician\_employee\_id"]), int(d["ticket\_id"]),

        d.get("start\_at"), d.get("end\_at"), d.get("notes"),

    )

    return {"id": wid}, 201

@bp.get("/by-ticket/<int:ticket\_id>")

def list\_work(ticket\_id:int):

    return repo.list\_by\_ticket(ticket\_id)

database:

database.py:

import sqlite3

from contextlib import contextmanager

from pathlib import Path

from .schema import SCHEMA

DB\_PATH = Path(\_\_file\_\_).resolve().parent.parent / "db.sqlite"

def \_row\_factory(cursor, row):

    return {col[0]: row[idx] for idx, col in enumerate(cursor.description)}

@contextmanager

def get\_conn():

    conn = sqlite3.connect(DB\_PATH)

    conn.row\_factory = \_row\_factory

    conn.execute("PRAGMA foreign\_keys = ON;")

    try:

        yield conn

        conn.commit()

    finally:

        conn.close()

def init\_db() -> None:

    with get\_conn() as c:

        c.executescript(SCHEMA)

schema.py:

# database/schema.py

SCHEMA = """

PRAGMA foreign\_keys = ON;

-- Users

CREATE TABLE IF NOT EXISTS user (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  username TEXT NOT NULL UNIQUE,

  password\_hash TEXT NOT NULL,

  email TEXT NOT NULL,

  name TEXT

);

-- Employee category

CREATE TABLE IF NOT EXISTS employee\_category (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL UNIQUE,

  description TEXT

);

-- Employees (Administrator/Operator/Technician via employee\_type)

CREATE TABLE IF NOT EXISTS employee (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  email TEXT NOT NULL,

  employee\_type TEXT NOT NULL CHECK (employee\_type IN ('ADMINISTRATOR','OPERATOR','TECHNICIAN')),

  employee\_category\_id INTEGER,

  user\_id INTEGER,

  FOREIGN KEY (employee\_category\_id) REFERENCES employee\_category(id),

  FOREIGN KEY (user\_id) REFERENCES user(id)

);

-- Clients

CREATE TABLE IF NOT EXISTS client (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  email TEXT NOT NULL,

  phone TEXT,

  address TEXT

);

-- Service taxonomy

CREATE TABLE IF NOT EXISTS category (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  description TEXT

);

CREATE TABLE IF NOT EXISTS service (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  description TEXT,

  category\_id INTEGER,

  FOREIGN KEY (category\_id) REFERENCES category(id)

);

-- Equipment

CREATE TABLE IF NOT EXISTS equipment\_category (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  description TEXT

);

CREATE TABLE IF NOT EXISTS equipment (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  serial TEXT NOT NULL,

  equipment\_category\_id INTEGER,

  FOREIGN KEY (equipment\_category\_id) REFERENCES equipment\_category(id)

);

-- Teams

CREATE TABLE IF NOT EXISTS team (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  name TEXT NOT NULL,

  description TEXT

);

CREATE TABLE IF NOT EXISTS team\_employee (

  team\_id INTEGER NOT NULL,

  employee\_id INTEGER NOT NULL,

  PRIMARY KEY (team\_id, employee\_id),

  FOREIGN KEY (team\_id) REFERENCES team(id) ON DELETE CASCADE,

  FOREIGN KEY (employee\_id) REFERENCES employee(id) ON DELETE CASCADE

);

-- Tickets

CREATE TABLE IF NOT EXISTS ticket (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  client\_id INTEGER NOT NULL,

  reporter\_employee\_id INTEGER,

  team\_id INTEGER,

  state TEXT NOT NULL CHECK (state IN ('OPEN','IN\_PROGRESS','RESOLVED','CLOSED')),

  created\_at TEXT NOT NULL,

  FOREIGN KEY (client\_id) REFERENCES client(id),

  FOREIGN KEY (reporter\_employee\_id) REFERENCES employee(id),

  FOREIGN KEY (team\_id) REFERENCES team(id)

);

CREATE TABLE IF NOT EXISTS ticket\_employee (

  ticket\_id INTEGER NOT NULL,

  employee\_id INTEGER NOT NULL,

  PRIMARY KEY (ticket\_id, employee\_id),

  FOREIGN KEY (ticket\_id) REFERENCES ticket(id) ON DELETE CASCADE,

  FOREIGN KEY (employee\_id) REFERENCES employee(id) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS ticket\_service (

  ticket\_id INTEGER NOT NULL,

  service\_id INTEGER NOT NULL,

  PRIMARY KEY (ticket\_id, service\_id),

  FOREIGN KEY (ticket\_id) REFERENCES ticket(id) ON DELETE CASCADE,

  FOREIGN KEY (service\_id) REFERENCES service(id) ON DELETE CASCADE

);

-- Incidents

CREATE TABLE IF NOT EXISTS incident (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  ticket\_id INTEGER NOT NULL,

  description TEXT NOT NULL,

  date TEXT NOT NULL,

  FOREIGN KEY (ticket\_id) REFERENCES ticket(id)

);

-- Messages

CREATE TABLE IF NOT EXISTS message (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  ticket\_id INTEGER NOT NULL,

  sender\_employee\_id INTEGER NOT NULL,

  date TEXT NOT NULL,

  content TEXT NOT NULL,

  FOREIGN KEY (ticket\_id) REFERENCES ticket(id),

  FOREIGN KEY (sender\_employee\_id) REFERENCES employee(id)

);

-- Work logs

CREATE TABLE IF NOT EXISTS worklog (

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  technician\_employee\_id INTEGER NOT NULL,

  ticket\_id INTEGER NOT NULL,

  start\_at TEXT,

  end\_at TEXT,

  notes TEXT,

  FOREIGN KEY (technician\_employee\_id) REFERENCES employee(id),

  FOREIGN KEY (ticket\_id) REFERENCES ticket(id)

);

-- Helpful indexes

CREATE INDEX IF NOT EXISTS idx\_employee\_type ON employee(employee\_type);

CREATE INDEX IF NOT EXISTS idx\_ticket\_client ON ticket(client\_id);

CREATE INDEX IF NOT EXISTS idx\_ticket\_state ON ticket(state);

CREATE INDEX IF NOT EXISTS idx\_incident\_ticket ON incident(ticket\_id);

CREATE INDEX IF NOT EXISTS idx\_message\_ticket ON message(ticket\_id);

"""

repositories:

base.py:

from typing import Any, Iterable

from database.database import get\_conn

class BaseRepo:

    def fetch\_one(self, sql: str, params: Iterable[Any] = ()):

        with get\_conn() as c:

            return c.execute(sql, params).fetchone()

    def fetch\_all(self, sql: str, params: Iterable[Any] = ()):

        with get\_conn() as c:

            return c.execute(sql, params).fetchall()

    def execute(self, sql: str, params: Iterable[Any] = ()):

        with get\_conn() as c:

            cur = c.execute(sql, params)

            return cur.lastrowid

category\_repo.py:

from .base import BaseRepo

class CategoryRepo(BaseRepo):

    def create(self, name: str, description: str | None = None) -> int:

        return self.execute("INSERT INTO category (name,description) VALUES (?,?)", (name, description))

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM category WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM category ORDER BY name")

    def update(self, id: int, name: str, description: str | None):

        return self.execute("UPDATE category SET name=?, description=? WHERE id=?", (name, description, id))

    def delete(self, id: int): return self.execute("DELETE FROM category WHERE id=?", (id,))

client\_repo.py:

from .base import BaseRepo

class ClientRepo(BaseRepo):

    def create(self, name: str, email: str, phone: str | None = None, address: str | None = None) -> int:

        return self.execute(

            "INSERT INTO client (name,email,phone,address) VALUES (?,?,?,?)", (name, email, phone, address)

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM client WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM client ORDER BY id DESC")

    def update(self, id: int, \*\*fields):

        cols = []

        vals = []

        for k, v in fields.items():

            cols.append(f"{k}=?")

            vals.append(v)

        vals.append(id)

        return self.execute(f"UPDATE client SET {', '.join(cols)} WHERE id=?", vals)

    def delete(self, id: int): return self.execute("DELETE FROM client WHERE id=?", (id,))

employee\_repo.py:

from .base import BaseRepo

class EmployeeRepo(BaseRepo):

    def create(self, name: str, email: str, employee\_type: str,

               employee\_category\_id: int | None = None, user\_id: int | None = None) -> int:

        return self.execute(

            "INSERT INTO employee (name,email,employee\_type,employee\_category\_id,user\_id) VALUES (?,?,?,?,?)",

            (name, email, employee\_type, employee\_category\_id, user\_id),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM employee WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM employee ORDER BY id DESC")

    def update(self, id: int, \*\*fields):

        cols = []

        vals = []

        for k, v in fields.items():

            cols.append(f"{k}=?")

            vals.append(v)

        vals.append(id)

        return self.execute(f"UPDATE employee SET {', '.join(cols)} WHERE id=?", vals)

    def delete(self, id: int): return self.execute("DELETE FROM employee WHERE id=?", (id,))

equipment\_category\_repo.py:

from .base import BaseRepo

class EquipmentCategoryRepo(BaseRepo):

    def create(self, name: str, description: str | None = None) -> int:

        return self.execute("INSERT INTO equipment\_category (name,description) VALUES (?,?)", (name, description))

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM equipment\_category WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM equipment\_category ORDER BY name")

    def update(self, id: int, name: str, description: str | None):

        return self.execute("UPDATE equipment\_category SET name=?, description=? WHERE id=?", (name, description, id))

    def delete(self, id: int): return self.execute("DELETE FROM equipment\_category WHERE id=?", (id,))

equipment\_repo.py:

from .base import BaseRepo

class EquipmentRepo(BaseRepo):

    def create(self, name: str, serial: str, equipment\_category\_id: int | None = None) -> int:

        return self.execute(

            "INSERT INTO equipment (name,serial,equipment\_category\_id) VALUES (?,?,?)",

            (name, serial, equipment\_category\_id),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM equipment WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM equipment ORDER BY id DESC")

    def update(self, id: int, \*\*fields):

        cols, vals = [], []

        for k, v in fields.items():

            cols.append(f"{k}=?"); vals.append(v)

        vals.append(id)

        return self.execute(f"UPDATE equipment SET {', '.join(cols)} WHERE id=?", vals)

    def delete(self, id: int): return self.execute("DELETE FROM equipment WHERE id=?", (id,))

incident\_repo.py:

from datetime import datetime

from .base import BaseRepo

class IncidentRepo(BaseRepo):

    def create(self, ticket\_id: int, description: str, date: str | None = None) -> int:

        date = date or datetime.utcnow().isoformat(timespec="seconds")

        return self.execute("INSERT INTO incident (ticket\_id,description,date) VALUES (?,?,?)",

                            (ticket\_id, description, date))

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM incident WHERE id=?", (id,))

    def list\_by\_ticket(self, ticket\_id: int):

        return self.fetch\_all("SELECT \* FROM incident WHERE ticket\_id=? ORDER BY id", (ticket\_id,))

messege\_repo.py:

from datetime import datetime

from .base import BaseRepo

class MessageRepo(BaseRepo):

    def create(self, ticket\_id: int, sender\_employee\_id: int, content: str, date: str | None = None) -> int:

        date = date or datetime.utcnow().isoformat(timespec="seconds")

        return self.execute(

            "INSERT INTO message (ticket\_id,sender\_employee\_id,date,content) VALUES (?,?,?,?)",

            (ticket\_id, sender\_employee\_id, date, content),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM message WHERE id=?", (id,))

    def list\_by\_ticket(self, ticket\_id: int):

        return self.fetch\_all("SELECT \* FROM message WHERE ticket\_id=? ORDER BY id", (ticket\_id,))

service\_repo.py:

from .base import BaseRepo

class ServiceRepo(BaseRepo):

    def create(self, name: str, description: str | None = None, category\_id: int | None = None) -> int:

        return self.execute(

            "INSERT INTO service (name,description,category\_id) VALUES (?,?,?)",

            (name, description, category\_id),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM service WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM service ORDER BY id DESC")

    def update(self, id: int, \*\*fields):

        cols, vals = [], []

        for k, v in fields.items():

            cols.append(f"{k}=?"); vals.append(v)

        vals.append(id)

        return self.execute(f"UPDATE service SET {', '.join(cols)} WHERE id=?", vals)

    def delete(self, id: int): return self.execute("DELETE FROM service WHERE id=?", (id,))

team\_repo.py:

from .base import BaseRepo

class TeamRepo(BaseRepo):

    def create(self, name: str, description: str | None = None) -> int:

        return self.execute("INSERT INTO team (name,description) VALUES (?,?)", (name, description))

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM team WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM team ORDER BY id DESC")

    def update(self, id: int, name: str, description: str | None):

        return self.execute("UPDATE team SET name=?, description=? WHERE id=?", (name, description, id))

    def delete(self, id: int): return self.execute("DELETE FROM team WHERE id=?", (id,))

    def add\_member(self, team\_id: int, employee\_id: int):

        return self.execute(

            "INSERT OR IGNORE INTO team\_employee (team\_id,employee\_id) VALUES (?,?)",

            (team\_id, employee\_id),

        )

ticket\_repo.py:

from datetime import datetime

from .base import BaseRepo

class TicketRepo(BaseRepo):

    def create(self, client\_id: int, reporter\_employee\_id: int | None,

               team\_id: int | None, state: str = "OPEN",

               created\_at: str | None = None) -> int:

        created\_at = created\_at or datetime.utcnow().isoformat(timespec="seconds")

        return self.execute(

            """INSERT INTO ticket (client\_id, reporter\_employee\_id, team\_id, state, created\_at)

               VALUES (?,?,?,?,?)""",

            (client\_id, reporter\_employee\_id, team\_id, state, created\_at),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM ticket WHERE id=?", (id,))

    def list(self): return self.fetch\_all("SELECT \* FROM ticket ORDER BY id DESC")

    def set\_state(self, id: int, state: str): return self.execute("UPDATE ticket SET state=? WHERE id=?", (state, id))

    # links

    def add\_employee(self, ticket\_id: int, employee\_id: int):

        return self.execute("INSERT OR IGNORE INTO ticket\_employee (ticket\_id,employee\_id) VALUES (?,?)",

                            (ticket\_id, employee\_id))

    def add\_service(self, ticket\_id: int, service\_id: int):

        return self.execute("INSERT OR IGNORE INTO ticket\_service (ticket\_id,service\_id) VALUES (?,?)", (ticket\_id, service\_id))

user\_repo.py:

from .base import BaseRepo

class UserRepo(BaseRepo):

    def create(self, username: str, password\_hash: str, email: str, name: str | None = None) -> int:

        return self.execute(

            "INSERT INTO user (username,password\_hash,email,name) VALUES (?,?,?,?)",

            (username, password\_hash, email, name),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM user WHERE id=?", (id,))

    def get\_by\_username(self, username: str): return self.fetch\_one("SELECT \* FROM user WHERE username=?", (username,))

    def list(self): return self.fetch\_all("SELECT \* FROM user ORDER BY id DESC")

    def delete(self, id: int): return self.execute("DELETE FROM user WHERE id=?", (id,))

work\_repo.py:

from .base import BaseRepo

class WorkRepo(BaseRepo):

    def create(self, technician\_employee\_id: int, ticket\_id: int,

               start\_at: str | None, end\_at: str | None, notes: str | None) -> int:

        return self.execute(

            "INSERT INTO worklog (technician\_employee\_id,ticket\_id,start\_at,end\_at,notes) VALUES (?,?,?,?,?)",

            (technician\_employee\_id, ticket\_id, start\_at, end\_at, notes),

        )

    def get(self, id: int): return self.fetch\_one("SELECT \* FROM worklog WHERE id=?", (id,))

    def list\_by\_ticket(self, ticket\_id: int):

        return self.fetch\_all("SELECT \* FROM worklog WHERE ticket\_id=? ORDER BY id", (ticket\_id,))

archivos “libres”:

app.py:

from flask import Flask

from flask\_cors import CORS

from flasgger import Swagger

from database.database import init\_db

import yaml, os

init\_db()

app = Flask(\_\_name\_\_)

CORS(app)

with open(os.path.join("docs", "swagger.yaml"), "r", encoding="utf-8") as f:

    template = yaml.safe\_load(f)

Swagger(app, template=template)

from controllers.auth\_controller import bp as auth\_bp

from controllers.client\_controller import bp as client\_bp

from controllers.employee\_controller import bp as employee\_bp

from controllers.category\_controller import bp as category\_bp

from controllers.service\_controller import bp as service\_bp

from controllers.equipment\_category\_controller import bp as ecat\_bp

from controllers.equipment\_controller import bp as equip\_bp

from controllers.team\_controller import bp as team\_bp

from controllers.ticket\_controller import bp as ticket\_bp

from controllers.incident\_controller import bp as incident\_bp

from controllers.message\_controller import bp as message\_bp

from controllers.work\_controller import bp as work\_bp

app.register\_blueprint(auth\_bp)

app.register\_blueprint(client\_bp)

app.register\_blueprint(employee\_bp)

app.register\_blueprint(category\_bp)

app.register\_blueprint(service\_bp)

app.register\_blueprint(ecat\_bp)

app.register\_blueprint(equip\_bp)

app.register\_blueprint(team\_bp)

app.register\_blueprint(ticket\_bp)

app.register\_blueprint(incident\_bp)

app.register\_blueprint(message\_bp)

app.register\_blueprint(work\_bp)

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(host="0.0.0.0", port=5000, debug=True)

managers.py:

import sqlite3

from models import Incidente, Ticket

from database import DB\_NAME

from datetime import datetime

class IncidenteManager:

    def \_\_init\_\_(self):

        self.conn = sqlite3.connect(DB\_NAME, check\_same\_thread=False)

        self.conn.row\_factory = sqlite3.Row

    def listar(self):

        rows = self.conn.execute("SELECT \* FROM incidentes").fetchall()

        return [Incidente(\*\*dict(r)) for r in rows]

    def crear(self, descripcion, tipo):

        cur = self.conn.cursor()

        cur.execute("INSERT INTO incidentes (descripcion, tipo) VALUES (?, ?)", (descripcion, tipo))

        self.conn.commit()

        return Incidente(cur.lastrowid, descripcion, tipo)

    def obtener(self, incidente\_id):

        row = self.conn.execute("SELECT \* FROM incidentes WHERE id=?", (incidente\_id,)).fetchone()

        return Incidente(\*\*dict(row)) if row else None

class TicketManager:

    def \_\_init\_\_(self):

        self.conn = sqlite3.connect(DB\_NAME, check\_same\_thread=False)

        self.conn.row\_factory = sqlite3.Row

    def listar(self):

        rows = self.conn.execute("SELECT \* FROM tickets").fetchall()

        return [Ticket(\*\*dict(r)) for r in rows]

    def crear(self, cliente, servicio, incidente\_id):

        fecha = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        cur = self.conn.cursor()

        cur.execute("""

            INSERT INTO tickets (cliente, servicio, incidente\_id, estado, fecha\_creacion)

            VALUES (?, ?, ?, ?, ?)

        """, (cliente, servicio, incidente\_id, "Abierto", fecha))

        self.conn.commit()

        return Ticket(cur.lastrowid, cliente, servicio, incidente\_id, "Abierto", fecha)

    def obtener(self, ticket\_id):

        row = self.conn.execute("SELECT \* FROM tickets WHERE id=?", (ticket\_id,)).fetchone()

        return Ticket(\*\*dict(row)) if row else None

    def cerrar(self, ticket\_id):

        fecha\_cierre = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        self.conn.execute("UPDATE tickets SET estado=?, fecha\_cierre=? WHERE id=?",

                          ("Cerrado", fecha\_cierre, ticket\_id))

        self.conn.commit()

        return self.obtener(ticket\_id)

models.py:

from dataclasses import dataclass

from datetime import datetime

from typing import Optional

@dataclass

class Incidente:

    id: int

    descripcion: str

    tipo: str

@dataclass

class Ticket:

    id: int

    cliente: str       # hardcodeado

    servicio: str      # hardcodeado

    incidente\_id: int

    estado: str = "Abierto"

    fecha\_creacion: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    fecha\_cierre: Optional[str] = None