**EMPLOYEE CONTACT MANAGEMENT SYSTEM (FastAPI + MongoDB and CRUD operations)**

**Project Overview:**

In this Project we are going to look how an organization saves the employee details(Profesional (department,position) and emergency contact details) in database(MONGO DB)by using FASTAPI. Here, we are using CRUD(Create,Read,Update,Delete)operations to create a user employee Create Endpoint was used, to check the employee details Read Endpoint was used, to update the employee details(i.e the employee promotion updation or employee contact details information updation) If an employee gets off from the organization to remove the employee details from the organization Delete operation was used. Lets dive deep into know more.

**Prerequisites**

Python 3.12+ installed (Download Python)

Code editor (VS Code recommended) with Python extension

MongoDB compass

Basic understanding of programming concepts(FastAPIs,CRUD operations)

To start making a code the first most rule is to make a perfect folder structure.Lets start making folder structure

main.py

config.py

README.md

requirement.txt

utilis.py

routes.py

crud.py

models.py

database.py

**main.py:**

* It is the **entry point** for the application
* Creates the FastAPI instance and includes routers
* This file is run by Uvicorn.

from fastapi import FastAPI

from app.routes import router as employee\_router

app = FastAPI(title="Employee Contact Management System")

app.include\_router(employee\_router)

**config.py:**

* Stores **configurations** like database URI and database name.
* Loads environment variables from .env.
* Exposes MONGO\_URI and DB\_NAME for other modules.

import os

from dotenv import load\_dotenv

load\_dotenv()

MONGO\_URI = os.getenv("MONGO\_URI", "mongodb+srv://akmudumbai:5xoOh4i261MnkmFa@myprojects.ylji0m2.mongodb.net/")

DB\_NAME = os.getenv("DB\_NAME", "employee\_db")

**database.py:**

* Creates the MongoClient using MONGO\_URI
* For Atlas: enables TLS and points to certifi CA bundle.
* Exposes a handle to the employees collection so CRUD can use it.

from pymongo import MongoClient

from app.config import MONGO\_URI, DB\_NAME

client = MongoClient(

    MONGO\_URI,

    tls=True,

    tlsCAFile=certifi.where()

)

db = client[DB\_NAME]

employee\_collection = db["employees"]

**models.py:**

* It makes a data in a particular model
* It validates the data
* Defines Employee (POST/PUT input) and UpdateEmployee (partial updates).
* Ensures fields like email are valid before hitting DB.

from pydantic import BaseModel, EmailStr

from typing import Optional

class Employee(BaseModel):

    name: str

    email: EmailStr

    phone: str

    department: Optional[str] = None

    position: Optional[str] = None

class UpdateEmployee(BaseModel):

    name: Optional[str] = None

    email: Optional[EmailStr] = None

    phone: Optional[str] = None

    department: Optional[str] = None

    position: Optional[str] = None

**crud.py**

**C** Create

**R** Read

**U** Update

**D** Delete

* It Implements
  + - create\_employee,
    - get\_all\_employees,
    - get\_employee,
    - update\_employee,
    - delete\_employee.
* Serializes Mongo ObjectId to string ("id").

from app.database import employee\_collection

from bson import ObjectId

def employee\_serializer(employee) -> dict:

    return {

        "id": str(employee["\_id"]),

        "name": employee["name"],

        "email": employee["email"],

        "phone": employee["phone"],

        "department": employee.get("department"),

        "position": employee.get("position")

    }

# CREATE

def create\_employee(employee: dict):

    result = employee\_collection.insert\_one(employee)

    return str(result.inserted\_id)

# READ ALL

def get\_all\_employees():

    employees = employee\_collection.find()

    return [employee\_serializer(emp) for emp in employees]

# READ ONE

def get\_employee(emp\_id: str):

    employee = employee\_collection.find\_one({"\_id": ObjectId(emp\_id)})

    return employee\_serializer(employee) if employee else None

# UPDATE

def update\_employee(emp\_id: str, data: dict):

    employee\_collection.update\_one({"\_id": ObjectId(emp\_id)}, {"$set": data})

    return get\_employee(emp\_id)

# DELETE

def delete\_employee(emp\_id: str):

    result = employee\_collection.delete\_one({"\_id": ObjectId(emp\_id)})

    return result.deleted\_count > 0

**routes.py:**

* Maps URLs and methods to CRUD functions.
* Raises meaningful HTTP errors (404, 400, etc.).
* Returns consistent JSON using a response formatter

from fastapi import APIRouter, HTTPException

from app.models import Employee, UpdateEmployee

from app import crud

router = APIRouter()

@router.post("/employees")

def create(employee: Employee):

    emp\_id = crud.create\_employee(employee.dict())

    return {"message": "Employee created successfully", "id": emp\_id}

@router.get("/employees")

def get\_all():

    return crud.get\_all\_employees()

@router.get("/employees/{emp\_id}")

def get\_one(emp\_id: str):

    employee = crud.get\_employee(emp\_id)

    if not employee:

        raise HTTPException(status\_code=404, detail="Employee not found")

    return employee

@router.put("/employees/{emp\_id}")

def update(emp\_id: str, employee: UpdateEmployee):

    updated = crud.update\_employee(emp\_id, {k: v for k, v in employee.dict().items() if v is not None})

    if not updated:

        raise HTTPException(status\_code=404, detail="Employee not found")

    return {"message": "Employee updated successfully", "employee": updated}

@router.delete("/employees/{emp\_id}")

def delete(emp\_id: str):

    success = crud.delete\_employee(emp\_id)

    if not success:

        raise HTTPException(status\_code=404, detail="Employee not found")

    return {"message": "Employee deleted successfully"}

**utilis.py:**

* validate\_object id: checks emp\_id is a valid Mongo ObjectId.
* Object id\_to\_str: converts ObjectId → string for responses.
* response\_formatter: standard response envelope { success, message, data }.

**Interview Answers**

1. **Why FastAPI instead of Django/Flask?**
2. **Why MongoDB instead of MySQL/PostgreSQL?**
3. **What are Pydantic models in FastAPI?**
4. **What is APIRouter?**
5. **Difference between JSON and BSON?**
6. **Why MongoDB uses BSON instead of JSON**?

**Interview Answers**

1. **Why FastAPI instead of Django/Flask?**  
   **ans:** FastAPI is faster, asynchronous, has automatic Swagger docs, and uses Pydantic for validation. It’s lightweight compared to Django and more modern than Flask.
2. **Why MongoDB instead of MySQL/PostgreSQL?**

Ans: MongoDB is NoSQL, document-based, and schema-flexible. It’s good for projects where data structure may change (like employee details). It also scales easily for large datasets.

1. **What are Pydantic models in FastAPI?**

Ans: Pydantic models define data schemas with validation. Example: ensuring email is valid or phone is an integer before saving.

1. **What is APIRouter?**

Ans: APIRouter helps in modularizing code by grouping routes (e.g., employee routes separate from auth routes).

1. **Difference between JSON and BSON?**  
   Ans: JSON = text-based, BSON = binary (MongoDB’s internal format, supports extra data types like ObjectId, Date).
2. **Why MongoDB uses BSON instead of JSON**?

Ans: MongoDB uses BSON instead of JSON because BSON is binary, faster to parse, and supports additional data types like ObjectId and Date, which JSON doesn’t. It’s optimized for database storage and indexing, while JSON is mainly for human readability and APIs