Restaurant

Acest proiect isi propune sa modeleze conceptul minimalistic al unui restaurant sub forma unei baze de date SQL. Ca abordare, am luat in calcul abstractizarea urmatoarelor entitati:

- Client: persoana ce este servita in cadrul restaurantului, poate plasa o comanda si este servita de un angajat.
- Angajat: persoana ce serveste potentiali clienti
- Comanda: plasata de un client, ce contine unul sau mai multe produse
- Produs: ce poate fi servit prin intermediul a 0 sau mai multe comenzi
- Meniu: calup de produse compuse ce poate fi servit independent

Aceasta baza de date va fi folosita de o aplicatie desktop si va simula cateva scenarii posibile ce pot aparea intr-un astfel de restaurant. Aceasta va implementa tiparul MVC si va folosi ORM-uri peste intrarile ce sunt introduse cu ajutorul sintaxei SQL. View-ul va fi unul CLI si va avea posibilitatea sa poata fi expus si printr-un API prin intermediul unui serviciu web.

Scenariu

Un **client** ocupa o masa in restaurant si este in curand servit de un **angajat**. Angajatul preia **comanda** clientului, aceasta fiind formata din mai multe **produse** si un **meniu** special. Clientul este servit, consuma, plateste si paraseste restaurantul.

Structura

Aplicatia va fi structurata sub forma unei biblioteci (pachet de module) in Python capabila sa manipuleze baza de date in cauza, sa expuna functii utile pentru manipularea intrarilor si chiar sa ofere un API, astfel mutand nivelul interactiunilor in spatiul web. Alaturi de aceasta biblioteca, vor mai fi dezvoltate si scripturi utile care sa acceseze intr-o maniera CLI functionalitatile ei, reusind astfel sa simuleze scenarii ca cel descris mai sus.

Baza de date

In aceasta sectiune vom intra in detaliu cu privire la clasele, tabelele si datele cu care vom alcatui baza de date.

In fisierul *restaurant.zargo* avem diagrama de clase ce descrie la nivel abstract compozitia bazei de date ce va fi creata, cat si relatiile dintre acestea. In fisierul *restaurant.sql* avem codul SQL ce executa instructiunile necesare crearii bazei de date si adaugarii de intrari initiale (de test).

Alegerea tabelelor

Entitatile descrise in scenariu au condus la concretizarea urmatoarelor tabele:

CUSTOMERS - contine clientii restaurantului, acestia expun campuri ca:

- CustomerId (Integer)
- CustomerName (String)
- Phone (String)
- *Address (String)*
- *Email (String)*

ORDERS - comenzile plasate de clienti:

- ProductId (Integer)

- OrderDate (String)
- TotalPrice (Double)
- *TotalKcal (Integer)*

PRODUCTS - produsele ce pot fi comandate prin intermediul unei comenzi:

- ProductId (Integer)
- ProductName (String)
- Price (Double)
- *Kcal (Integer)*

EMPLOYEES - angajatii restaurantului (chelneri) ce vor servi clientii:

- EmployeeId (Integer)
- EmployeeName (String)
- Salary (Integer)
- *Age (Integer)*

MENUS - meniuri pe post de ansamblu de produse:

- MenuId (Integer)
- MenuName (String)
- Description (String)
- Price (Double)

ORDERPRODUCT - solutionarea relatiei many-to-many intre ORDER si PRODUCT:

- OrderId (Integer)
- ProductId (Integer)

Unde avem urmatoarea asociere de tipuri de date:

- Integer -> *NUMBER*
- String -> *VARCHAR(N)*
- Double -> *DOUBLE PRECISION*

Campurile in italic de mai sus sunt optionale (pot fi nule).

Relatii & Constrangeri

1. **Customer-Order**: relatie *one-to-one*, deoarece fiecarui client i se cuvine o comanda.
   - PKs: CUSTOMERS.CustomerId, ORDERS.OrderId
   - FKs: CUSTOMERS.OrderId
2. **Customer-Employee**: relatie *many-to-one*, unde un angajat serveste mai multi clienti.
   - PKs: CUSTOMERS.CustomerId, EMPLOYEES.EmployeeId
   - FKs: CUSTOMERS.EmployeeId
3. **Order-Product**: relatie *many-to-many*, unde mai multe comenzi pot contine mai multe produse.
   - PKs: ORDERPRODUCT.OrderId, ORDERPRODUCT.ProductId
   - FKs: ORDERPRODUCT.OrderId, ORDERPRODUCT.ProductId