Podstawy Baz Danych

“Restauracja”

Wspomaganie działalności firmy świadczącej usługi gastronomiczne dla klientów indywidualnych oraz firm

Filip Kitka

Mateusz Łopaciński

Mateusz Wronka

Spis treści

1. Użytkownicy systemu

1. Administrator
2. System
3. Menedżer restauracji
4. Pracownik restauracji
5. Klient indywidualny
6. Firma

2. Funkcje systemu

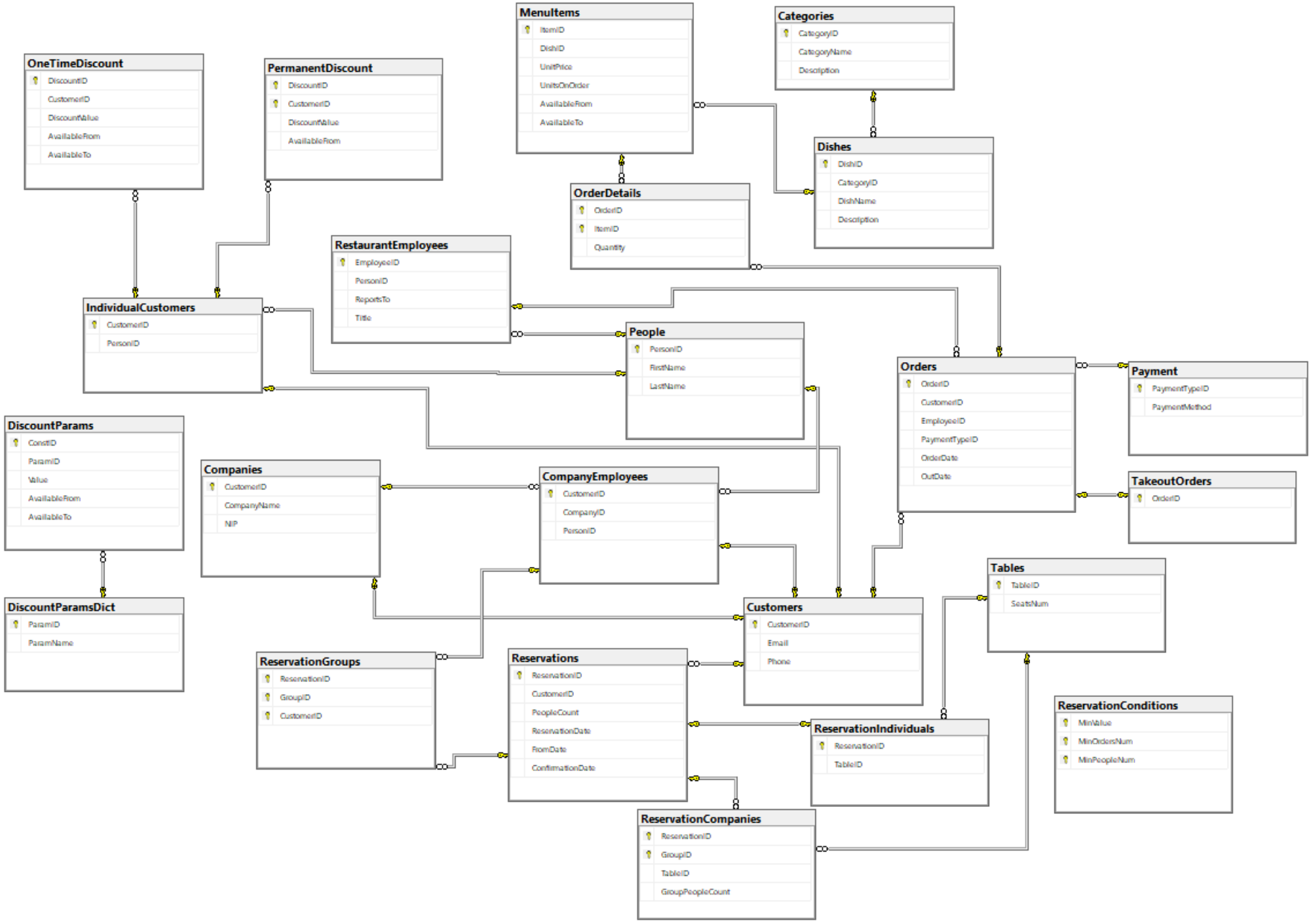
Administrator ma dostęp do wszystkich funkcji

- tworzenie backupów bazy danych
  Administrator
- tworzenie i edycja kont pracowników i menedżera
  Administrator
- generowanie raportów z działalności restauracji
  Menedżer restauracji
  - możliwość wybrania przedziału czasu
- automatyczna zmiana połowy pozycji w menu
  System
  - wybór najstarszych zamówień z obecnego menu, brakujących do połowy zmienionych w okresie co najmniej 2 ostatnich tygodni
    System
  - możliwość edycji/dodania pozycji do proponowanej listy dań do zmiany
    Menedżer restauracji
- generowanie statystyk zamówienia dla konkretnych klientów i firm
  Menedżer restauracji
- generowanie ilości zamówionych owoców morza (w odpowiednim przedziale czasowym)
  Menedżer restauracji
- dodawanie pozycji do menu
  Menedżer restauracji
  - nazwa pozycji
  - cena pozycji

    - limit ilościowy do którego można zamawiać daną pozycję
    - ustalenie czasu dostępności
- edycja pozycji w menu
  Menedżer restauracji
- potwierdzenie rezerwacji stolika ze wskazaniem stolika
  Pracownik restauracji
- dostęp do aktualnych zamówień
  Pracownik restauracji
- potwierdzenie realizacji zamówienia
  Pracownik restauracji
- informacje o rezerwacji stolika (kto zarezerwował)
  Pracownik restauracji
- złożenie zamówienia
  Klient indywidualny, Firma
    - wybór z menu
    - zamówienie na miejscu
    - zamówienie na wynos z wyprzedzeniem
        - formularz WWW
        - wybór daty i godziny odbioru
    - zamówienie na wynos na miejscu
    - możliwość skorzystania z jednorazowej zniżki (w przypadku braku skorzystania ze zniżki, naliczona zostaje domyślna zniżka)
      tylko klient indywidualny
    - wybór z menu owoców morza
        - w dniach czwartek, piątek, sobota
        - zamówienie musi zostać złożone do poniedziałku poprzedzającego zamówienie
- rezerwacja stolika ze złożeniem zamówienia
  Klient indywidualny, Firma
    - płatność przed odebraniem zamówienia lub po odebraniu
- generowanie raportów dotyczących zamówień oraz rabatów dla klienta
  Klient indywidualny, Firma
    - wyświetlanie brakującej kwoty do otrzymania kolejnej zniżki

- generowanie statystyk zamówień
  Klient indywidualny, Firma
- generowanie informacji o swoich oczekujących zamówieniach
  Klient indywidualny, Firma
- generowanie informacji o swoich rezerwacjach
  Klient indywidualny, Firma
- założenie konta klienta indywidualnego
  Klient indywidualny
    - dane osobowe
    - dane kontaktowe
- wystawienie faktury dla zamówienia lub faktury zbiorczej raz na miesiąc
  Firma
- założenie konta firmy
  Firma
    - dane firmy
    - pracownicy firmy (firma podaje dane pracowników, których konta mają zostać utworzone)

3. Schemat bazy danych



4. Tabele

1. MenuItems

Tabela przechowująca pozycje w menu oraz daty, kiedy były dostępne.

a. **PK** ItemID (int, NOT NULL) - ID dania w Menu

b. **FK(Dishes)** DishID (int, NOT NULL) - ID dania w Dishes

c. UnitPrice (money, NOT NULL) - cena dania

d. UnitsOnOrder (int, NOT NULL) - ile razy danie może być zamówione

e. AvailableFrom (datetime, NOT NULL) - od kiedy danie jest dostępne w menu

f. AvailableTo (datetime N) - do kiedy danie było dostępne w menu

Warunki integralnościowe

1. UnitsOnOrder jest liczbą dodatnią (`CONSTRAINT CHK_UnitsOnOrder CHECK (UnitsOnOrder >= 0)`)
2. AvailableFrom domyślnie jest datą obecną(`DEFAULT GETDATE()`)
3. AvailableTo nie jest wcześniej niż AvailableFrom(`CONSTRAINT CHK_AvailableMenuItems CHECK (AvailableFrom < ISNULL(AvailableTo, GETDATE()))`

Kod generujący tabelę

```
CREATE TABLE MenuItems (

   ItemID int  NOT NULL IDENTITY(1, 1),

   DishID int  NOT NULL,

   UnitPrice money  NOT NULL,

   UnitsOnOrder int  NOT NULL,

   AvailableFrom datetime  NOT NULL DEFAULT GETDATE(),

   AvailableTo datetime  NULL,

   CONSTRAINT MenuItems_pk PRIMARY KEY (ItemID),

   CONSTRAINT CHK_UnitsOnOrder CHECK (UnitsOnOrder >= 0),

   CONSTRAINT CHK_AvailableMenuItems CHECK (AvailableFrom <
ISNULL(AvailableTo, GETDATE()))

);


ALTER TABLE MenuItems ADD CONSTRAINT Dishes_Menu

   FOREIGN KEY (DishID)
```

```
    REFERENCES Dishes (DishID);
```

2. Dishes

W tej tabeli przechowujemy wszystkie dania, również owoce morza oraz przypisane im kategorie i nazwy. Owoce morza rozróżniamy, przy pomocy kategorii, do której należą.

a. **PK** DishID (int, NOT NULL) - ID dania
b. **FK(Categories)** CategoryID (int, NOT NULL) - ID kategorii
c. DishName (varchar(40), NOT NULL) - nazwa dania
d. Description (varchar(100) N) - opis dania

Warunki integralnościowe

1. DishName jest unikalne (`CONSTRAINT DishName_ak UNIQUE(DishName)`)

Kod generujący tabelę

```
CREATE TABLE Dishes (
   DishID int  NOT NULL IDENTITY(1, 1),
   CategoryID int  NOT NULL,
   DishName varchar(40)  NOT NULL,
   Description varchar(100)  NULL,
   CONSTRAINT Dishes_pk PRIMARY KEY (DishID),
   CONSTRAINT DishName_ak UNIQUE(DishName)
);


ALTER TABLE Dishes ADD CONSTRAINT Dishes_Categories
   FOREIGN KEY (CategoryID)
   REFERENCES Categories (CategoryID);
```

3. Categories

Tabela, służąca do przechowywania nazw kategorii produktów oraz ich opisów.

a. **PK** CategoryID (int, NOT NULL) - ID kategorii
b. CategoryName (varchar(20), NOT NULL) - nazwa kategorii
c. Description (varchar(100) N) - opis kategorii

Warunki integralnościowe

1. CategoryName jest unikalne (`CONSTRAINT CategoryName_ak UNIQUE(CategoryName)`)

Kod generujący tabelę

```
CREATE TABLE Categories (
    CategoryID int  NOT NULL IDENTITY(1, 1),
    CategoryName varchar(20)  NOT NULL,
    Description varchar(100)  NULL,
    CONSTRAINT Categories_pk PRIMARY KEY (CategoryID),
    CONSTRAINT CategoryName_ak UNIQUE(CategoryName)
);
```

4. Orders

Tabela przechowująca wszystkie zamówienia oraz ich dane

a. **PK** OrderID (int, NOT NULL) - ID zamówienia

b. **FK(Customers)** CustomerID (int, NULL) - ID klienta składającego zamówienie

c. **FK(RestaurantEmployees)** EmployeeID (int, NOT NULL) - ID pracownika restauracji przyjmującego zamówienie

d. **FK(Payment)** PaymentTypeID (int, NULL) - ID metody płatności, prowadzące do tabeli słownikowej Payment

e. OrderDate (datetime, NOT NULL) - data złożenia zamówienia

f. OutDate (datetime N) - data odebrania zamówienia przez klienta

Warunki integralnościowe

1. OrderDate domyślnie jest datą obecną (`DEFAULT GETDATE()`)
2. OrderDate jest wcześniejszą datą niż OutDate (`CONSTRAINT CHK_Date CHECK (OrderDate <= ISNULL(OutDate, GETDATE()))`)

Kod generujący tabelę

```
CREATE TABLE Orders (
    OrderID int  NOT NULL IDENTITY(1, 1),
    CustomerID int  NULL,
    EmployeeID int  NOT NULL,
```

```
    PaymentTypeID int  NULL,

    OrderDate datetime  NOT NULL DEFAULT GETDATE(),

    OutDate datetime  NULL,

    CONSTRAINT Orders_pk PRIMARY KEY (OrderID),

    CONSTRAINT CHK_Date CHECK (OrderDate <= ISNULL(OutDate,
GETDATE()))

);

ALTER TABLE Orders ADD CONSTRAINT Orders_Customers

    FOREIGN KEY (CustomerID)

    REFERENCES Customers (CustomerID);


ALTER TABLE Orders ADD CONSTRAINT Orders_Payment

    FOREIGN KEY (PaymentTypeID)

    REFERENCES Payment (PaymentTypeID);


ALTER TABLE Orders ADD CONSTRAINT Orders_RestaurantEmployees

    FOREIGN KEY (EmployeeID)

    REFERENCES RestaurantEmployees (EmployeeID);
```

5. OrderDetails

Tabela przechowująca szczegóły zamówień.

a. **PK FK(Orders)** OrderID (int, NOT NULL) - ID zamówienia

b. **PK FK(MenuItems)** ItemID (int, NOT NULL) - ID zamówionego w danym zamówieniu dania z menu

c. Quantity (int, NOT NULL) - liczba zamówionych dań

Warunki integralnościowe

1. Quantity jest liczbą dodatnią (`CONSTRAINT CHK_Quantity CHECK (Quantity > 0)`)

Kod generujący tabelę

```
CREATE TABLE OrderDetails (

    OrderID int  NOT NULL,
```

```
    ItemID int  NOT NULL,
    Quantity int  NOT NULL,
    CONSTRAINT OrderDetails_pk PRIMARY KEY (OrderID,ItemID),
    CONSTRAINT CHK_Quantity CHECK (Quantity > 0)
);

ALTER TABLE OrderDetails ADD CONSTRAINT MenuItems_OrderDetails
    FOREIGN KEY (ItemID)
    REFERENCES MenuItems (ItemID);

ALTER TABLE OrderDetails ADD CONSTRAINT OrderDetails_Orders
    FOREIGN KEY (OrderID)
    REFERENCES Orders (OrderID);
```

6. TakeoutOrders

Tabela przechowująca ID zamówień na wynos.

a. **PK FK(Orders)** OrderID (int, NOT NULL) - ID zamówienia

Kod generujący tabelę

```
CREATE TABLE TakeoutOrders (
    OrderID int  NOT NULL,
    CONSTRAINT TakeoutOrders_pk PRIMARY KEY (OrderID)
);

ALTER TABLE TakeoutOrders ADD CONSTRAINT TakeoutOrders_Orders
    FOREIGN KEY (OrderID)
    REFERENCES Orders (OrderID);
```

7. Payment

Tabela słownikowa przechowująca dostępne formy płatności

a. **PK** PaymentTypeID (int, NOT NULL) - ID formy płatności

b. PaymentMethod (varchar(20), NOT NULL) - nazwa formy płatności

Warunki integralnościowe

1. PaymentMethod jest unikalne (`CONSTRAINT PaymentMethod_ak UNIQUE (PaymentMethod)`)

Kod generujący tabelę

```
CREATE TABLE Payment (
   PaymentTypeID int  NOT NULL IDENTITY(1, 1),
   PaymentMethod varchar(20)  NOT NULL,
   CONSTRAINT Payment_pk PRIMARY KEY (PaymentTypeID),
   CONSTRAINT PaymentMethod_ak UNIQUE (PaymentMethod)
);
```

8. DiscountParamsDict

Tabela słownikowa przechowująca nazwy parametrów zniżek

a. **PK** ParamID (int, NOT NULL) - ID parametru
b. ParamName (varchar(2), NOT NULL) - nazwa parametru (Z1,R1,...)

Warunki integralnościowe

1. ParamName jest unikalne(`CONSTRAINT ParamName_ak UNIQUE (ParamName))`

Kod generujący tabelę

```
CREATE TABLE DiscountParamsDict (
   ParamID int  NOT NULL IDENTITY(1, 1),
   ParamName varchar(2)  NOT NULL,
   CONSTRAINT DiscountParamsDict_pk PRIMARY KEY  (ParamID),
   CONSTRAINT ParamName_ak UNIQUE (ParamName)
);
```

9. DiscountParams

Tabela przechowująca wartości parametrów zniżek wraz z datami, kiedy te parametry były aktualne

a. **PK** ConstID (int, NOT NULL) - ID parametru w danym okresie

b. **FK(DiscountParamsDict)** ParamID (int, NOT NULL) - ID parametru w tabeli DiscountParamsDict
c. Value (int, NOT NULL) - wartość parametru
d. AvailableFrom (datetime, NOT NULL) - od kiedy parametr miał daną wartość
e. AvailableTo (datetime N) - do kiedy parametr miał daną wartość

Warunki integralnościowe

1. Value jest liczbą dodatnią (`CONSTRAINT CHK_ParamValue CHECK (Value > 0)`)
2. AvailableFrom jest domyślnie obecną datą (`DEFAULT GETDATE()`)
3. AvailableFrom jest wcześniej niż AvailableTo (`CONSTRAINT CHK_ParamDate CHECK (AvailableFrom <= ISNULL(AvailableTo, GETDATE()))`)

Kod generujący tabelę

```
CREATE TABLE DiscountParams (

   ConstID int  NOT NULL IDENTITY(1, 1),

   ParamID int  NOT NULL,

   Value int  NOT NULL,

   AvailableFrom datetime  NOT NULL DEFAULT GETDATE(),

   AvailableTo datetime  NULL,

   CONSTRAINT DiscountParams_pk PRIMARY KEY NONCLUSTERED (ConstID),

   CONSTRAINT CHK_ParamValue CHECK (Value > 0),

   CONSTRAINT CHK_ParamDate CHECK (AvailableFrom <=

ISNULL(AvailableTo, GETDATE()))

);


ALTER TABLE DiscountParams ADD CONSTRAINT

DiscountParams_DiscountParamsDict

   FOREIGN KEY (ParamID)

   REFERENCES DiscountParamsDict (ParamID);
```

10. OneTimeDiscount

Tabela przechowująca informacje o tym, jakie jednorazowe zniżki są lub były dostępne dla konkretnych klientów oraz daty określające czas, kiedy zniżka była używana

a. **PK** DiscountID (int, NOT NULL) - ID zniżki, która była dostępna w danym czasie dla klienta
b. **FK(IndividualCustomers)** CustomerID (int, NOT NULL) - ID klienta, który korzystał ze zniżki
c. DiscountValue (int, NOT NULL) - wartość zniżki
d. AvailableFrom (datetime, NOT NULL) - kiedy zniżka zaczęła być używana
e. AvailableTo (datetime) - do kiedy zniżka była ważna

Warunki integralnościowe

1. DiscountValue jest wartością między 0 a 100 (oznacza procent) (`CONSTRAINT CHK_DiscountValueOneTimeDiscount CHECK (DiscountValue >= 0 AND DiscountValue <= 100)`)
2. AvailableFrom jest domyślnie datą obecną(`DEFAULT GETDATE()`)
3. AvailableFrom jest wcześniej niż AvailableTo (`CONSTRAINT CHK_AvailableOneTimeDiscount CHECK (AvailableFrom < ISNULL(AvailableTo, GETDATE()))`)

Kod generujący tabelę

```
CREATE TABLE OneTimeDiscount (

   DiscountID int  NOT NULL IDENTITY(1, 1),

   CustomerID int  NOT NULL,

   DiscountValue int  NOT NULL,

   AvailableFrom datetime  NOT NULL DEFAULT GETDATE(),

   AvailableTo datetime  NULL,

   CONSTRAINT OneTimeDiscount_pk PRIMARY KEY NONCLUSTERED
(DiscountID),

   CONSTRAINT CHK_DiscountValueOneTimeDiscount CHECK (DiscountValue
>= 0 AND DiscountValue <= 100),

   CONSTRAINT CHK_AvailableOneTimeDiscount CHECK (AvailableFrom <
ISNULL(AvailableTo, GETDATE()))

);
```

```
ALTER TABLE OneTimeDiscount ADD CONSTRAINT
OneTimeDiscount_IndividualCustomers

   FOREIGN KEY (CustomerID)

   REFERENCES IndividualCustomers (CustomerID);
```

11. PermanentDiscount

Tabela przechowująca informacje o tym, którzy klienci mają dostępną permanentną zniżkę, datę od kiedy jest ona dostępna, oraz jej wartość

a. **PK FK(IndividualCustomers)** CustomerID (int, NOT NULL) - ID klienta

b. DiscountValue (int, NOT NULL) - wartość zniżki

c. AvailableFrom (datetime, NOT NULL) - od kiedy zniżka była dostępna dla klienta

Warunki integralnościowe

1. AvailableFrom jest domyślnie obecną datą (`DEFAULT GETDATE()`)
2. DiscountValue jest wartością między 0 a 100 (oznacza procent) (`CONSTRAINT CHK_DiscountValuePermanentDiscount CHECK (DiscountValue >= 0 AND DiscountValue <= 100)`)

Kod generujący tabelę

```
CREATE TABLE PermanentDiscount (

   DiscountID int  NOT NULL IDENTITY(1, 1),

   CustomerID int  NOT NULL,

   DiscountValue int  NOT NULL,

   AvailableFrom datetime  NOT NULL DEFAULT GETDATE(),

   CONSTRAINT PermanentDiscount_pk PRIMARY KEY NONCLUSTERED
(CustomerID, DiscountID),

   CONSTRAINT CHK_DiscountValuePermanentDiscount CHECK
(DiscountValue >= 0 AND DiscountValue <= 100)

);



ALTER TABLE PermanentDiscount ADD CONSTRAINT
PermanentDiscount_IndividualCustomers

   FOREIGN KEY (CustomerID)
```

```
    REFERENCES IndividualCustomers (CustomerID);
```

12. RestaurantEmployees

Dane pracowników restauracji oraz informacje o ich przełożonych.

a. **PK** EmployeeID (int, NOT NULL) - ID pracownika restauracji
b. **FK(People)** PersonID (int, NOT NULL) - ID danych pracownika w tabeli People
c. ReportsTo (int N) - ID przełożonego pracownika
d. Title (varchar(20), NOT NULL) - tytuł pracownika

Kod generujący tabelę

```
CREATE TABLE RestaurantEmployees (
   EmployeeID int  NOT NULL IDENTITY(1, 1),
   PersonID int  NOT NULL,
   ReportsTo int  NULL,
   Title varchar(20)  NOT NULL,
   CONSTRAINT RestaurantEmployees_pk PRIMARY KEY (EmployeeID)
);


ALTER TABLE RestaurantEmployees ADD CONSTRAINT
People_RestaurantEmployees
   FOREIGN KEY (PersonID)
   REFERENCES People (PersonID);
```

13. People

Dane osobowe klientów oraz pracowników.

a. **PK** PersonID (int, NOT NULL) - ID osoby
b. FirstName (varchar(20), NOT NULL) - imię
c. LastName (varchar(20), NOT NULL) - nazwisko

Warunki integralnościowe

1. FirstName jest skapitalizowane (`CONSTRAINT CHK_FirstName CHECK (SUBSTRING(FirstName, 1, 1) = UPPER(SUBSTRING(FirstName, 1, 1)))`)
2. LastName jest skapitalizowane (`CONSTRAINT CHK_LastName CHECK (SUBSTRING(LastName, 1, 1) = UPPER(SUBSTRING(LastName, 1, 1)))`)

Kod generujący tabelę

```
CREATE TABLE People (

   PersonID int  NOT NULL IDENTITY(1, 1),

   FirstName varchar(20)  NOT NULL,

   LastName varchar(20)  NOT NULL,

   CONSTRAINT People_pk PRIMARY KEY (PersonID),

   CONSTRAINT CHK_FirstName CHECK (SUBSTRING(FirstName, 1, 1) =
UPPER(SUBSTRING(FirstName, 1, 1))),

   CONSTRAINT CHK_LastName CHECK (SUBSTRING(LastName, 1, 1) =
UPPER(SUBSTRING(LastName, 1, 1)))

);
```

14. Customers

Tabela przechowująca wszystkich zarejestrowanych klientów restauracji oraz ich dane kontaktowe.

a. **PK** CustomerID (int, NOT NULL) - ID klienta

b. Email (varchar(40), NOT NULL) - email klienta

c. Phone (varchar(15), NOT NULL) - numer telefonu klienta

Warunki integralnościowe

1. Phone jest w postaci np. +48555444333 lub 555444333 (`(Phone LIKE '+' + REPLICATE('[0-9]', 11) OR Phone LIKE REPLICATE('[0-9]', 9))`)
2. Email jest w postaci <nazwa>@<pełna_nazwa_domeny> (`CONSTRAINT CHK_Email CHECK (Email LIKE '%_@_%._%')`)

Kod generujący tabelę

```
CREATE TABLE Customers (

   CustomerID int  NOT NULL IDENTITY(1, 1),

   Email varchar(40)  NOT NULL,

   Phone varchar(12)  NOT NULL,

   CONSTRAINT Customers_pk PRIMARY KEY (CustomerID),
```

```
    CONSTRAINT Email_ak UNIQUE (Email),

    CONSTRAINT Phone_ak UNIQUE (Phone),

    CONSTRAINT CHK_Phone CHECK (Phone LIKE '+' + REPLICATE('[0-9]',
                11) OR Phone LIKE REPLICATE('[0-9]', 9)),

    CONSTRAINT CHK_Email CHECK (Email LIKE '%_@_%._%')

);
```

15. IndividualCustomers

Tabela łącząca klientów indywidualnych z ich danymi osobowymi, umieszczonymi w tabeli People oraz z tabelą Customers, zawierającą wszystkich klientów.

a. **PK FK(Customers)** CustomerID (int, NOT NULL) - ID klienta

b. **FK(People)** PersonID (int, NOT NULL) - ID danych klienta w tabeli People

Kod generujący tabelę

```
CREATE TABLE IndividualCustomers (

    CustomerID int  NOT NULL,

    PersonID int  NOT NULL,

    CONSTRAINT IndividualCustomers_pk PRIMARY KEY (CustomerID)

);


ALTER TABLE IndividualCustomers ADD CONSTRAINT
IndividualCustomers_Customers

    FOREIGN KEY (CustomerID)

    REFERENCES Customers (CustomerID);


ALTER TABLE IndividualCustomers ADD CONSTRAINT
IndividualCustomers_People

    FOREIGN KEY (PersonID)

    REFERENCES People (PersonID);
```

16. Companies

Tabela przechowująca dane firm, będących klientami restauracji.

a. **PK FK(Customers)** CustomerID (int, NOT NULL) - ID klienta

b. CompanyName (varchar(30), NOT NULL) - nazwa firmy

c. NIP (varchar(11), NOT NULL) - NIP firmy

Warunki integralnościowe

1. NIP jest w postaci 10 lub 11 cyfr (`CONSTRAINT ReservationCompanies_pk PRIMARY KEY (ReservationID,GroupID)`)

Kod generujący tabelę

```
CREATE TABLE Companies (
    CustomerID int  NOT NULL,
    CompanyName varchar(30)  NOT NULL,
    NIP varchar(11)  NOT NULL,
    CONSTRAINT Companies_pk PRIMARY KEY (CustomerID),
    CONSTRAINT CHK_NIP CHECK (NIP LIKE  REPLICATE('[0-9]', 10) OR NIP
LIKE REPLICATE('[0-9]', 11))
);


ALTER TABLE Companies ADD CONSTRAINT Customers_Companies
    FOREIGN KEY (CustomerID)
    REFERENCES Customers (CustomerID);
```

17. CompanyEmployees

Tabela łącząca pracowników firm, będącymi klientami restauracji, z danymi firmy oraz danymi osobowymi pracownika.

a. **PK FK(Customers)** CustomerID (int, NOT NULL) - ID klienta

b. **FK(Companies)** CompanyID (int, NOT NULL) - ID firmy, której dany klient jest pracownikiem

c. **FK(People)** PersonID (int, NOT NULL) - ID danych klienta w tabeli People

Kod generujący tabelę

```
CREATE TABLE CompanyEmployees (
   CustomerID int  NOT NULL,
   CompanyID int  NOT NULL,
   PersonID int  NOT NULL,
   CONSTRAINT CompanyEmployees_pk PRIMARY KEY (CustomerID)
);

ALTER TABLE CompanyEmployees ADD CONSTRAINT
CompanyEmployees_Companies
   FOREIGN KEY (CompanyID)
   REFERENCES Companies (CustomerID);

ALTER TABLE CompanyEmployees ADD CONSTRAINT CompanyEmployees_People
   FOREIGN KEY (PersonID)
   REFERENCES People (PersonID);

ALTER TABLE CompanyEmployees ADD CONSTRAINT
Customers_CompanyEmployees
   FOREIGN KEY (CustomerID)
   REFERENCES Customers (CustomerID);
```

18. ReservationIndividuals

Tabela łącząca zatwierdzone rezerwacje, złożone przez klientów indywidualnych, ze stolikami.

a. **PK FK** ReservationID (int, NOT NULL) - ID rezerwacji

b. **FK** TableID (int, NOT NULL) - ID zarezerwowanego stolika

Kod generujący tabelę

```
CREATE TABLE ReservationIndividuals (
   ReservationID int  NOT NULL,
   TableID int  NOT NULL,
   CONSTRAINT ReservationIndividuals_pk PRIMARY KEY
(ReservationID)
);
```

```
ALTER TABLE ReservationIndividuals ADD CONSTRAINT
Reservations_ReservationIndividuals

  FOREIGN KEY (ReservationID)

  REFERENCES Reservations (ReservationID);


ALTER TABLE ReservationIndividuals ADD CONSTRAINT
Tables_ReservationIndividuals

   FOREIGN KEY (TableID)

   REFERENCES Tables (TableID);
```

19. Reservations

Tabela przechowująca dane, dotyczące rezerwacji.

a. **PK** ReservationID (int, NOT NULL) - ID rezerwacji

b. **FK(Customers)** CustomerID (int, NOT NULL) - ID klienta składającego rezerwację

c. PeopleCount (int, NOT NULL) - liczba osób na którą została złożona rezerwacja

d. ReservationDate (datetime, NOT NULL) - data złożenia rezerwacji przez klienta

e. FromDate (datetime, NOT NULL) - data, na którą została złożona rezerwacja

f. ConfirmationDate (datetime N) - data zatwierdzenia rezerwacji przez pracownika restauracji (null oznacza, że rezerwacja jeszcze nie została potwierdzona)

Warunki integralnościowe

1. ReservationDate jest wcześniej niż FromDate(`CONSTRAINT CHK_ReservationDate CHECK (ReservationDate <= FromDate)`)
2. FromDate jest wcześniej niż ConfirmationDate(`CONSTRAINT CHK_FromDate CHECK (FromDate >= ISNULL(ConfirmationDate, GETDATE()))`)

Kod generujący tabelę

```
CREATE TABLE Reservations (

   ReservationID int  NOT NULL IDENTITY(1, 1),
```

```
    CustomerID int  NOT NULL,

    PeopleCount int  NOT NULL,

    ReservationDate datetime  NOT NULL DEFAULT GETDATE(),

    FromDate datetime  NOT NULL,

    ConfirmationDate datetime  NULL,

    CONSTRAINT Reservations_pk PRIMARY KEY (ReservationID),

    CONSTRAINT CHK_ReservationDate CHECK (ReservationDate <=
FromDate),

    CONSTRAINT CHK_FromDate CHECK (FromDate >=
ISNULL(ConfirmationDate, GETDATE()))

);


ALTER TABLE Reservations ADD CONSTRAINT Reservations_Customers

    FOREIGN KEY (CustomerID)

    REFERENCES Customers (CustomerID);
```

20. ReservationGroups

Tabela przechowująca grupy pracowników firmy, która złożyła rezerwację, którzy mają siedzieć przy jednym stoliku

a. **PK FK(Reservations)** ReservationID (int, NOT NULL) - ID rezerwacji
b. **PK** GroupID (int, NOT NULL) - ID grupy
c. **PK FK(CompanyEmployees)** CustomerID (int, NOT NULL) - ID klienta, który jest pracownikiem firmy składającej rezerwację

Kod generujący tabelę

```
CREATE TABLE ReservationGroups (

    ReservationID int  NOT NULL,

    GroupID int  NOT NULL,

    CustomerID int  NOT NULL,

    CONSTRAINT ReservationGroups_pk PRIMARY KEY
(ReservationID,GroupID,CustomerID)

);


ALTER TABLE ReservationGroups ADD CONSTRAINT
Reservations_ReservationGroups
```

```
   FOREIGN KEY (ReservationID)
   REFERENCES Reservations (ReservationID);


ALTER TABLE ReservationGroups ADD CONSTRAINT
ReservationGroups_CompanyEmployees
   FOREIGN KEY (CustomerID)
   REFERENCES CompanyEmployees (CustomerID);
```

21. ReservationCompanies

Tabela przechowująca zatwierdzone rezerwacje firmowe, oraz łącząca grupy pracowników ze stolikami

a. **PK FK(Reservations)** ReservationID (int, NOT NULL) - ID rezerwacji
b. **PK** GroupID (int, NOT NULL) - ID grupy pracowników
c. **PK FK(Tables)** TableID (int, NOT NULL) - ID stolika
d. GroupPeopleCount (int, NOT NULL) - Liczba osób, wchodzących w skład grupy

Kod generujący tabelę

```
CREATE TABLE ReservationCompanies (
   ReservationID int  NOT NULL,
   GroupID int  NOT NULL,
   TableID int  NOT NULL,
   GroupPeopleCount int NOT NULL,
   CONSTRAINT ReservationCompanies_pk PRIMARY KEY
(ReservationID,GroupID)
);


ALTER TABLE ReservationCompanies ADD CONSTRAINT
ReservationCompanies_Tables
   FOREIGN KEY (TableID)
   REFERENCES Tables (TableID);


ALTER TABLE ReservationCompanies ADD CONSTRAINT
Reservations_ReservationCompanies
   FOREIGN KEY (ReservationID)
```

```
REFERENCES Reservations (ReservationID);
```

22. Tables

Tabela przechowująca informacje o stolikach.

a. **PK** TableID (int, NOT NULL) - ID stolika
b. SeatsNum (int, NOT NULL) - liczba miejsc przy stoliku

Warunki integralnościowe

1. SeatsNum musi być większe niż 0 (`CONSTRAINT CHK_SeatsNum CHECK (SeatsNum > 0)`)

Kod generujący tabelę

```
CREATE TABLE Tables (
   TableID int  NOT NULL IDENTITY(1, 1),
   SeatsNum int  NOT NULL,
   CONSTRAINT Tables_pk PRIMARY KEY (TableID),
   CONSTRAINT CHK_SeatsNum CHECK (SeatsNum > 0)
);
```

23. ReservationConditions

Tabela przechowująca minimalną ilość zamówień oraz minimalną wartość zamówienia wymagane do złożenia rezerwacji.

a. **PK** MinValue(money, NOT NULL) - minimalna wartość złożonych zamówień
b. **PK** MinOrderNum(int, NOT NULL) - minimalna liczba zamówień
c. **PK** MinPeopleNum(int, NOT NULL) - minimalna liczba osób, na które można złożyć rezerwację

Warunki integralnościowe

1. MinOrdersNum jest liczbą dodatnią (`CONSTRAINT CHK_MinOrdersNum CHECK (MinOrdersNum > 0)`)
2. MinPeopleNum jest liczbą dodatnią (`CONSTRAINT CHK_MinPeopleNum CHECK (MinPeopleNum > 0)`)

Kod generujący tabelę

```
CREATE TABLE ReservationConditions (
   MinValue money  NOT NULL,
   MinOrdersNum int  NOT NULL,
   MinPeopleNum int  NOT NULL,
   CONSTRAINT ReservationConditions_pk PRIMARY KEY
(MinValue,MinOrdersNum,MinPeopleNum),
   CONSTRAINT CHK_MinOrdersNum CHECK (MinOrdersNum > 0),
   CONSTRAINT CHK_MinPeopleNum CHECK (MinPeopleNum > 0)
);
```

5. Widoki

1. **CurrentMenuView** - Pokazuje aktualne menu

```
CREATE VIEW CurrentMenuView
AS SELECT * FROM MenuItems
WHERE AvailableTo IS NULL OR AvailableTo > GETDATE();
```

2. **PendingReservationsView** - Pokazuje rezerwacje, które nie zostały jeszcze zatwierdzone przez pracownika restauracji

```
CREATE VIEW PendingReservationsView
AS SELECT * FROM Reservations
WHERE ConfirmationDate IS NULL;
```

3. **IndividualsReservationsView** - Pokazuje rezerwacje złożone przez klientów indywidualnych

```
CREATE VIEW IndividualsReservationsView
AS SELECT r.ReservationID, r.CustomerID, ri.TableID, ri.PeopleCount,
r.ReservationDate FROM Reservations AS r
INNER JOIN ReservationIndividuals AS ri
ON ri.ReservationID = r.ReservationID;
```

4. **CurrentIndividualsReservationsView** - Pokazuje aktualne rezerwacje złożone przez klientów indywidualnych

```
CREATE VIEW CurrentIndividualsReservationsView
AS SELECT * FROM IndividualsReservationsView
```

```
WHERE DATEDIFF(DAY, r.ReservationDate, GETDATE()) = 0;
```

5. **IndividualsReservationsMonthlyReportView** - Pokazuje rezerwacje złożone przez klientów indywidualnych w ostatnim miesiącu

```
CREATE VIEW IndividualReservationsMonthlyReportView

AS SELECT * FROM IndividualsReservationsView

WHERE DATEDIFF(DAY, r.ReservationDate, GETDATE()) <= 30;
```

6. **IndividualsReservationsWeeklyReportView** - Pokazuje rezerwacje złożone przez klientów indywidualnych w ostatnim tygodniu

```
CREATE VIEW IndividualReservationsWeeklyReportView

AS SELECT * FROM IndividualsReservationsView

WHERE DATEDIFF(DAY, r.ReservationDate, GETDATE()) <= 7;
```

7. **CompaniesReservationsView** - Pokazuje wszystkie rezerwacje złożone przez firmy, wraz z przydziałem pracowników firmy do stolików

```
CREATE VIEW CompaniesReservationsView

AS SELECT r.ReservationID, r.CustomerID AS CompanyID, rg.CustomerID AS
CompanyEmployeeID, rc.TableID, r.ReservationDate, r.FromDate

FROM Reservations AS r

INNER JOIN ReservationGroups AS rg

ON rg.ReservationID = r.ReservationID

INNER JOIN ReservationCompanies AS rc

ON rc.GroupID = rg.GroupID AND rc.ReservationID = r.ReservationID;
```

8. **CurrentCompaniesReservationsView** - Pokazuje dzisiejsze rezerwacje złożone przez firmy

```
CREATE VIEW CurrentCompaniesReservationsView

AS SELECT *

FROM CurrentCompaniesReservationsView

WHERE DATEDIFF(DAY, ReservationDate, GETDATE()) = 0;
```

9. **CompaniesMonthlyReservationsReportView** - Pokazuje rezerwacje złożone przez firmy w ostatnim miesiącu

```
CREATE VIEW CompaniesMonthlyReservationsReportView
AS SELECT * FROM CompaniesReservationsView
WHERE DATEDIFF(DAY, ReservationDate, GETDATE()) <= 30;
```

10. **CompaniesWeeklyReservationsReportView** - Pokazuje rezerwacje złożone przez firmy w ostatnim tygodniu

```
CREATE VIEW CompaniesWeeklyReservationsReportView
AS SELECT * FROM CompaniesReservationsView
WHERE DATEDIFF(DAY, ReservationDate, GETDATE()) <= 7;
```

11. **DishPopularityView** - Pokazuje wszystkie dania według ich popularości (ile razy zostały zamówione)

```
CREATE VIEW DishPopularityView
AS SELECT d.DishID, d.DishName, SUM(od.Quantity) AS TotalQuantity
FROM Dishes AS d
INNER JOIN MenuItems AS mi
ON d.DishID = mi.DishID
INNER JOIN OrderDetails AS od
ON mi.ItemID = od.ItemID
GROUP BY d.DishID, d.DishName;
```

12. **DishIncomeView** - Pokazuje przychody ze sprzedaży każdego dania

```
CREATE VIEW DishIncomeView
AS SELECT mi.DishID, SUM(od.Quantity * mi.UnitPrice * (1 - (
   odtv.DiscountValue / 100)) AS TotalIncome
FROM MenuItems AS mi
INNER JOIN OrderDetails AS od
ON od.ItemID = mi.ItemID
INNER JOIN OrdersDiscountsTableView AS odtv
ON odtv.OrderID = od.OrderID
GROUP BY mi.DishID;
```

13. **SeafoodMenuView** - Pokazuje wszystkie dostępne dania z owoców morza

```
CREATE VIEW SeafoodMenuView
AS SELECT mi.ItemID, d.DishID, d.DishName
FROM MenuItems AS mi
INNER JOIN Dishes AS d
ON d.DishID=mi.DishID
INNER JOIN Categories AS c
ON d.CategoryID=c.CategoryID
WHERE c.CategoryName='Seafood';
```

14. **SeafoodWeekOrdersView** - Pokazuje wszystkie zamówienia owoców morza w ostatnim tygodniu

```
CREATE VIEW SeafoodWeekOrdersView
AS SELECT o.OrderID, od.ItemID, od.Quantity, o.OrderDate, o.OutDate
FROM SeafoodMenuView AS smv
INNER JOIN OrderDetails AS od
ON od.ItemID = smv.ItemID
INNER JOIN Orders AS o
ON o.OrderID = od.OrderID
WHERE DATEDIFF(DAY, o.OrderDate, GETDATE()) <= 7;
```

15. **PendingOrdersView** - Pokazuje wszystkie zamówienia które nie zostały jeszcze odebrane

```
CREATE VIEW PendingOrdersView
AS SELECT *
FROM Orders
WHERE OutDate IS NULL OR OutDate > GETDATE();
```

16. **IndividualCustomersView** - Pokazuje wszystkich klientów indywidualnych oraz ich imiona i nazwiska

```
CREATE VIEW IndividualCustomersView
AS SELECT c.CustomerID, p.FirstName, p.LastName, c.Email, c.Phone
```

```
FROM IndividualCustomers AS ic
INNER JOIN People AS p
ON ic.PersonID = p.PersonID
INNER JOIN Customers AS c
ON c.CustomerID = ic.CustomerID;
```

17. **CompanyEmployeesView** - Pokazuje wszystkich klientów będących pracownikami firm oraz ich imiona i nazwiska

```
CREATE VIEW CompanyEmployeesView
AS SELECT c.CustomerID, p.FirstName, p.LastName, c.Email, c.Phone,
co.CustomerID AS CompanyID, co.CompanyName
FROM CompanyEmployees AS ce
INNER JOIN Companies AS co
ON co.CustomerID = ce.CompanyID
INNER JOIN Customers AS cu
ON cu.CustomerID = ce.CustomerID
INNER JOIN People AS p
ON p.PersonID = ce.PersonID;
```

18. **TakeoutOrdersView** - Pokazuje wszystkie zamówienia na wynos

```
CREATE VIEW TakeoutOrdersView
AS SELECT *
FROM Orders
WHERE OrderID IN (
 SELECT OrderID FROM TakeoutOrders
);
```

19. **VacantTablesView** - Pokazuje wszystkie wolne stoliki

```
CREATE VIEW VacantTablesView
AS SELECT *
FROM Tables
WHERE TableID NOT IN (
 SELECT TableID
```

```
 FROM CurrentCompaniesReservationsView
 UNION
 SELECT TableID
 FROM CurrentIndividualsReservationsView
);
```

20. **CustomerOneTimeDiscountsView** - Pokazuje wszystkie jednorazowe zniżki które były dostępne dla konkretnych klientów

```
CREATE VIEW CustomerOneTimeDiscountsView
AS SELECT c.CustomerID, otd.DiscountValue, otd.AvailableFrom,
otd.AvailableTo
FROM Customers AS c
INNER JOIN IndividualCustomers AS ic
ON c.CustomerID=ic.CustomerID
INNER JOIN OneTimeDiscount AS otd
ON ic.CustomerID=ic.CustomerID;
```

21. **CustomerPermanentDiscountsView** - Pokazuje permanentne zniżki dostępne dla konkretnych klientów

```
CREATE VIEW CustomerPermanentDiscountsView
AS SELECT ic.CustomerID, pd.DiscountValue, pd.AvailableFrom
FROM IndividualCustomers AS ic
INNER JOIN PermanentDiscount AS pd
ON pd.CustomerID = ic.CustomerID;
```

22. **UnpaidOrdersView** - Pokazuje zamówienia, które nie zostały jeszcze opłacone

```
CREATE VIEW UnpaidOrdersView
AS SELECT OrderID, CustomerID, OrderDate
FROM Orders
WHERE PaymentTypeID IS NULL;
```

23. **CurrentOneTimeDiscountParamsView** - Pokazuje aktualne parametry jednorazowej zniżki

```
CREATE VIEW CurrentOneTimeDiscountParamsView
AS SELECT ParamName, Value, AvailableFrom
FROM DiscountParamsTable
WHERE ParamName IN ('K2', 'R2', 'D1')
 AND (AvailableTo IS NULL
 OR AvailableTo > GETDATE());
```

24. **CurrentPermanentDiscountParamsView** - Pokazuje aktualne parametry permanentnej zniżki

```
CREATE VIEW CurrentPermanentDiscountParamsView
AS SELECT ParamName, Value, AvailableFrom
FROM DiscountParamsTable
WHERE ParamName IN ('Z1', 'K1', 'R1')
 AND (AvailableTo IS NULL
 OR AvailableTo > GETDATE());
```

25. **IndividualCustomersOrdersView** - Pokazuje całkowite koszty zamówień złożonych przez klientów indywidualnych

```
CREATE VIEW IndividualCustomersOrdersView
AS SELECT
 o.CustomerID,
 SUM(od.Quantity * mi.UnitPrice * (1 - odtv.DiscountValue / 100)) AS
TotalPrice,
 o.OrderDate,
 o.OutDate
FROM IndividualCustomers AS ic
INNER JOIN Orders AS o
ON o.CustomerID = ic.CustomerID
INNER JOIN OrderDetails AS od
ON od.OrderID = o.OrderID
INNER JOIN MenuItems AS mi
ON mi.ItemID = od.ItemID
```

```
INNER JOIN OrdersDiscountsTableView AS odtv
ON odv.OrderID = o.OrderID
GROUP BY o.OrderID, o.OrderDate, o.OutDate, o.CustomerID;
```

26. **IndividualCustomersWeeklyReportOrderView** - Pokazuje dane zamówień złożone przez klientów indywidualnych w ostatnim tygodniu

```
CREATE VIEW CompanyCustomersWeeklyOrderReportView
AS SELECT * FROM CompanyCustomersOrdersView
WHERE DATEDIFF(DAY, OrderDate, GETDATE()) <= 7;
```

27. **IndividualCustomersMonthlyReportOrderView** - Pokazuje dane zamówień złożone przez klientów indywidualnych w ostatnim miesiącu

```
CREATE VIEW CompanyCustomersMonthlyOrderReportView
AS SELECT * FROM CompanyCustomersOrdersView
WHERE DATEDIFF(DAY, OrderDate, GETDATE()) <= 30 ;
```

28. **CompanyCustomersOrdersView** - Pokazuje całkowite koszty zamówień złożonych przez firmy

```
CREATE VIEW CompanyCustomersOrdersView
AS SELECT
 o.OrderID,
 o.CustomerID,
 SUM(od.Quantity * mi.UnitPrice) AS TotalPrice,
 o.OrderDate,
 o.OutDate
FROM Companies AS c
INNER JOIN Orders AS o
ON o.CustomerID = c.CustomerID
INNER JOIN OrderDetails AS od
ON od.OrderID = o.OrderID
INNER JOIN MenuItems AS mi
ON mi.ItemID = od.ItemID
GROUP BY o.OrderID, o.OrderDate, o.OutDate, o.CustomerID;
```

29. **CompanyCustomersMonthlyReportOrderView** - Pokazuje dane zamówień złożone przez firmy w ostatnim miesiącu

```
CREATE VIEW CompanyCustomersMonthlyOrderReportView
AS SELECT * FROM CompanyCustomersOrdersView
WHERE DATEDIFF(DAY, OrderDate, GETDATE()) <= 30 ;
```

30. **CompanyCustomersWeeklyReportOrderView** - Pokazuje dane zamówień złożone przez firmy w ostatnim tygodniu

```
CREATE VIEW CompanyCustomersWeeklyOrderReportView
AS SELECT * FROM CompanyCustomersOrdersView
WHERE DATEDIFF(DAY, OrderDate, GETDATE()) <= 7;
```

31. **CompanyEmployeesOrdersView** - Pokazuje całkowite koszty zamówień złożonych przez klientów będących pracownikami firm

```
CREATE VIEW CompanyEmployeesOrdersView
AS SELECT
 o.OrderID,
 o.CustomerID,
 SUM(od.Quantity * mi.UnitPrice) AS TotalPrice,
 o.OrderDate,
 o.OutDate
FROM CompanyEmployees AS ce
INNER JOIN Orders AS o
ON o.CustomerID = ce.CustomerID
INNER JOIN OrderDetails AS od
ON od.OrderID = o.OrderID
INNER JOIN MenuItems AS mi
ON mi.ItemID = od.ItemID
GROUP BY o.OrderID, o.OrderDate, o.OutDate, o.CustomerID;
```

32. **CompanyCustomersMonthlyReportOrderView** - Pokazuje dane zamówień złożone przez pracowników firm w ostatnim tygodniu

```
CREATE VIEW CompanyEmployeesMonthlyOrderReportView
```

```
 AS SELECT * FROM CompanyEmployeesOrdersView
 WHERE DATEDIFF(DAY, OrderDate, GETDATE()) <= 30 ;
```

33. CompanyCustomersWeeklyReportOrderView - Pokazuje dane zamówień złożone przez pracowników firm w ostatnim miesiącu

```
 CREATE VIEW CompanyEmployeesWeeklyOrderReportView
 AS SELECT * FROM CompanyEmployeesOrdersView
 WHERE DATEDIFF(DAY, OrderDate, GETDATE()) <= 7;
```

34. DiscountParamsTableView - Pokazuje parametry dla zniżek

```
CREATE VIEW DiscountParamsTableView AS (
 SELECT ParamName, Value, AvailableFrom, AvailableTo
 FROM DiscountParams AS dp
 INNER JOIN DiscountParamsDict AS dpd
 ON dpd.ParamID = dp.ParamID
);
```

35. OrdersDiscountsTableView - Pokazuje ID zamówień i zniżki które zostały do nich zastosowane

```
CREATE VIEW OrdersDiscountsTableView AS
   SELECT o.OrderID, o.CustomerID, o.OrderDate, o.OutDate, ISNULL(
     (SELECT TOP 1 otd.DiscountValue
     FROM OneTimeDiscount AS otd
     WHERE otd.CustomerID = o.CustomerID AND o.OrderDate >=
otd.AvailableFrom
     AND o.OrderDate <= ISNULL(otd.AvailableTo, GETDATE()))
   , ISNULL(
     (SELECT TOP 1 pd.DiscountValue
     FROM PermanentDiscount AS pd
     WHERE pd.CustomerID = o.CustomerID AND o.OrderDate >=
pd.AvailableFrom)
  ,0)) AS DiscountValue
 FROM Orders AS o;
```

36. **UsedDiscountsView** - Pokazuje ID klienta, wartość zniżki, typ zniżki, ID zamówienia i czasy składania zamówienia, dla zniżek które zostały zastosowane do zamówień

```
CREATE VIEW UsedDiscountsView
AS SELECT odtv.CustomerID, odtv.DiscountValue, (
 SELECT IIF(EXISTS(
    SELECT *
    FROM CustomerOneTimeDiscountsView AS cotdv
    WHERE cotdv.CustomerID = odtv.CustomerID
      AND cotdv.AvailableFrom <= odtv.OrderDate
      AND odtv.OrderDate <= cotdv.AvailableTo
), 'OTD', 'PD')) AS DiscountType
, odtv.OrderID, odtv.OrderDate, odtv.OutDate
FROM OrdersDiscountsTableView AS odtv
INNER JOIN IndividualCustomers AS ic
ON ic.CustomerID = odtv.CustomerID
INNER JOIN Customers AS c
ON c.CustomerID = odtv.CustomerID
WHERE odtv.DiscountValue > 0;
```

37. **DiscountsMonthlyNumberView** - Pokazuje ilość użytych zniżek co miesiąc

```
CREATE VIEW DiscountsMonthlyNumberView
AS SELECT YEAR(OrderDate) AS UsedYear, MONTH(OrderDate) AS UsedMonth,
DiscountType, COUNT(*) AS DiscountsNumber
FROM UsedDiscountsView
GROUP BY YEAR(OrderDate),MONTH(OrderDate),DiscountType;
```

38. **OneTimeDiscountsMonthlyNumberView** - Pokazuje ilość użytych zniżek jednorazowych co miesiąc

```
CREATE VIEW OneTimeDiscountsMonthlyNumberView
AS SELECT UsedYear, UsedMonth, DiscountsNumber
FROM DiscountsMonthlyNumberView
WHERE DiscountType = 'OTD';
```

39. **PermanentDiscountsMonthlyNumberView** - Pokazuje ilość użytych zniżek permanentnych co miesiąc

```
CREATE VIEW PermanentDiscountsMonthlyNumberView

AS SELECT UsedYear, UsedMonth, DiscountsNumber

FROM DiscountsMonthlyNumberView

WHERE DiscountType = 'PD';
```

40. **IndividualTableStatsView** - Pokazuje ile razy zarezerowany był dany stolik przez klientów indywidualnych

```
CREATE VIEW IndividualTableStatsView

AS SELECT YEAR(r.ReservationDate) AS Year, MONTH(r.ReservationDate) AS
Month, ri.TableID, COUNT(*) as IndividualReservations

FROM Reservations as r

INNER JOIN ReservationIndividuals as ri on
r.ReservationID=ri.ReservationID

GROUP BY YEAR(r.ReservationDate), MONTH(r.ReservationDate), ri.TableID;
```

41. **CompanyTableStatsView** - Pokazuje ile razy zarezerowany był dany stolik przez firmy

```
CREATE VIEW CompanyTableStatsView

AS SELECT YEAR(r.ReservationDate) AS Year, MONTH(r.ReservationDate) as
Month, rc.TableID, COUNT(DISTINCT r.ReservationID) as CompanyReservations

FROM Reservations as r

INNER JOIN ReservationCompanies as rc on r.ReservationID =
rc.ReservationID

GROUP BY YEAR(r.ReservationDate), MONTH(r.ReservationDate), rc.TableID;
```

42. **TimeOfDayOrdersNumView** - Pokazuje ile zamówień jest w sumie zamawianych w danych porach dnia

```
CREATE VIEW TimeOfDayOrdersNumView

AS SELECT TOP 1

(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE DATEPART(HOUR,OrderDate) between 7 and 12

GROUP BY DATEPART(HOUR, OrderDate) WITH ROLLUP ORDER BY 1 DESC) as
'Morning',
```

```
(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE DATEPART(HOUR,OrderDate) between 13 and 18

GROUP BY DATEPART(HOUR, OrderDate) WITH ROLLUP ORDER BY 1 DESC) as
'Afternoon',

(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE DATEPART(HOUR,OrderDate) between 19 and 24

GROUP BY DATEPART(HOUR, OrderDate) WITH ROLLUP ORDER BY 1 DESC) as
'Evening',

(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE DATEPART(HOUR,OrderDate) between 0 and 6

GROUP BY DATEPART(HOUR, OrderDate) WITH ROLLUP ORDER BY 1 DESC) as
'Night'

FROM Orders;
```

43. **SeasonsOrdersNumView** - Pokazuje ile zamówień jest w sumie zamawianych w danych porach roku

```
CREATE VIEW SeasonsOrdersNumView

AS SELECT TOP 1

(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE MONTH(OrderDate) in (12, 1, 2)

GROUP BY MONTH(OrderDate) WITH ROLLUP ORDER BY 1 DESC) as 'Winter',

(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE MONTH(OrderDate) in (3, 4, 5)

GROUP BY MONTH(OrderDate) WITH ROLLUP ORDER BY 1 DESC) as 'Spring',

(SELECT TOP 1 COUNT(OrderID)

FROM Orders

WHERE MONTH(OrderDate) in (6, 7, 8)

GROUP BY MONTH(OrderDate) WITH ROLLUP ORDER BY 1 DESC) as 'Summer',

(SELECT TOP 1 COUNT(OrderID)

FROM Orders
```

```
WHERE MONTH(OrderDate) in (9, 10, 11)
GROUP BY MONTH(OrderDate) WITH ROLLUP ORDER BY 1 DESC) as 'Autumn'
FROM Orders;
```

44. **ReservedTablesView** - Pokazuje daty rezerwacji konkretnych stolików

```
CREATE VIEW ReservedTablesView
AS SELECT ri.TableID, r.FromDate
FROM ReservationIndividuals as ri
INNER JOIN Reservations as r on ri.ReservationID = r.ReservationID
UNION
SELECT rc.TableID, r.FromDate
FROM ReservationCompanies as rc
INNER JOIN Reservations as r on rc.ReservationID = r.ReservationID;
```

45. **DishesOlderThanTwoWeeksView** - Pokazuje pozycje w menu (oprócz owoców morza) dodane ponad 2 tygodnie temu

```
CREATE VIEW DishesOlderThanTwoWeeksView
AS SELECT * FROM CurrentMenuView
WHERE DATEDIFF(DAY, AvailableFrom, GETDATE()) > 14
AND DishID NOT IN (SELECT DishID FROM SeafoodMenuView);
```

6. Procedury

1. **AddIndividualCustomer** - dodaje klienta indywidualnego

```
CREATE PROCEDURE AddIndividualCustomer

 @FirstName varchar(20),

 @LastName varchar(20),

 @Email varchar(40),

 @Phone varchar(15)

AS

BEGIN

 BEGIN TRY

   INSERT INTO People (FirstName, LastName) VALUES (@FirstName,
@LastName);

   DECLARE @PersonID int;

   SELECT @PersonID = SCOPE_IDENTITY();


   INSERT INTO Customers (Email, Phone) VALUES (@Email, @Phone);

   DECLARE @CustomerID int;

   SELECT @CustomerID = SCOPE_IDENTITY();


   INSERT INTO IndividualCustomers(CustomerID, PersonID) VALUES
(@CustomerID, @PersonID);

 END TRY

 BEGIN CATCH

   DELETE FROM Customers WHERE CustomerID = @CustomerID

   DELETE FROM People WHERE PersonID = @PersonID

   DELETE FROM IndividualCustomers WHERE CustomerID = @CustomerID

   DECLARE @errorMsg nvarchar(1024) = 'Error while inserting Individual
Customer: '

   + ERROR_MESSAGE();

   THROW 52000, @errorMsg, 1;

 END CATCH;

END;
```

2. **AddCompany** - dodaje klienta firmowego

```
CREATE PROCEDURE AddCompany
 @CompanyName varchar(30),
 @NIP varchar(15),
 @Email varchar(40),
 @Phone varchar(15)
AS
BEGIN
 BEGIN TRY
   INSERT INTO Customers (Email, Phone) VALUES (@Email, @Phone);
   DECLARE @CustomerID int;
   SELECT @CustomerID = SCOPE_IDENTITY();

   INSERT INTO Companies(CustomerID, CompanyName, NIP) VALUES
(@CustomerID, @CompanyName, @NIP);
 END TRY
 BEGIN CATCH
   DELETE FROM Customers WHERE CustomerID = @CustomerID
   DELETE FROM Companies WHERE CustomerID = @CustomerID
   DECLARE @errorMsg nvarchar(1024) = 'Error while inserting Company: '
   + ERROR_MESSAGE();
   THROW 52000, @errorMsg, 1;
 END CATCH;
END;
```

3. **AddCompanyEmployee** - dodaje pracownika firmy

```
CREATE PROCEDURE AddCompanyEmployee
 @FirstName varchar(20),
 @LastName varchar(20),
 @Email varchar(40),
 @Phone varchar(15),
 @CompanyID int
AS
```

```
BEGIN
 IF (NOT EXISTS(SELECT * FROM Companies WHERE CustomerID=@CompanyID))
 BEGIN
   DECLARE @errorMsg1 nvarchar(1024) = 'Company does not exist';
   THROW 52000, @errorMsg1, 1;
 END;
 BEGIN TRY
   INSERT INTO People (FirstName, LastName) VALUES (@FirstName,
@LastName);
   DECLARE @PersonID int;
   SELECT @PersonID = SCOPE_IDENTITY();

   INSERT INTO Customers (Email, Phone) VALUES (@Email, @Phone);
   DECLARE @CustomerID int;
   SELECT @CustomerID = SCOPE_IDENTITY();

   INSERT INTO CompanyEmployees(CustomerID, CompanyID, PersonID) VALUES
(@CustomerID, @CompanyID, @PersonID);
 END TRY
 BEGIN CATCH
   DELETE FROM Customers WHERE CustomerID = @CustomerID
   DELETE FROM People WHERE PersonID = @PersonID
   DELETE FROM CompanyEmployees WHERE CustomerID = @CustomerID
   DECLARE @errorMsg2 nvarchar(1024) = 'Error while inserting Company
Employee: '
   + ERROR_MESSAGE();
   THROW 52000, @errorMsg2, 1;
 END CATCH;
END;
```

4. **AddRestaurantEmployee** - dodaje pracownika restauracji

```
CREATE PROCEDURE AddRestaurantEmployee
 @FirstName varchar(20),
 @LastName varchar(20),
```

```
 @ReportsTo int,

 @Title varchar(20)

AS

BEGIN

 IF (NOT EXISTS(SELECT * FROM RestaurantEmployees WHERE
EmployeeID=@ReportsTo))

 BEGIN

   DECLARE @errorMsg1 nvarchar(1024) = 'Employee does not exist';

   THROW 52000, @errorMsg1, 1;

 END;

 BEGIN TRY

   INSERT INTO People (FirstName, LastName) VALUES (@FirstName,
@LastName);

   DECLARE @PersonID int;

   SELECT @PersonID = SCOPE_IDENTITY();


   INSERT INTO RestaurantEmployees (PersonID, ReportsTo, Title) VALUES
(@PersonID, @ReportsTo, @Title)

   DECLARE @EmployeeID int;

   SELECT @EmployeeID = SCOPE_IDENTITY();

 END TRY

 BEGIN CATCH

   DELETE FROM RestaurantEmployees WHERE EmployeeID = @EmployeeID

   DELETE FROM People WHERE PersonID = @PersonID

   DECLARE @errorMsg2 nvarchar(1024) = 'Error while inserting Restaurant
Employee: '

   + ERROR_MESSAGE();

   THROW 52000, @errorMsg2, 1;

 END CATCH;

END;
```

5. **AddCategory** - dodaje kategrię dań

```
CREATE PROCEDURE AddCategory

 @CategoryName varchar(20)

 @Description varchar(100) NULL
```

```
AS

BEGIN

 BEGIN TRY

   INSERT INTO Categories (CategoryName, Description) VALUES
(@CategoryName, @Description);

 END TRY

 BEGIN CATCH

   DECLARE @errorMsg nvarchar(1024) = 'Error while inserting Category: '

   + ERROR_MESSAGE();

   THROW 52000, @errorMsg, 1;

 END CATCH;

END;
```

6. **AddDish**-dodaje nowe danie

```
CREATE PROCEDURE AddDish

 @CategoryID int,

 @DishName varchar(40),

 @Description varchar(100) NULL

AS

BEGIN

 IF (NOT EXISTS(SELECT * FROM Categories WHERE CategoryID=@CategoryID))

 BEGIN

   DECLARE @errorMsg1 nvarchar(1024) = 'Category does not exist';

   THROW 52000, @errorMsg1, 1;

 END;


 BEGIN TRY

   INSERT INTO Dishes (CategoryID, DishName, Description) VALUES
(@CategoryID, @DishName, @Description);

 END TRY

 BEGIN CATCH

   DECLARE @errorMsg2 nvarchar(1024) = 'Error while inserting Dish: '

   + ERROR_MESSAGE();

   THROW 52000, @errorMsg2, 1;
```

```
 END CATCH
END;
```

7. **AddMenuItem** - dodaje danie do menu

```
CREATE PROCEDURE AddMenuItem
 @DishID int,
 @UnitPrice money,
 @UnitsOnOrder int,
 @AvailableFrom datetime,
 @AvailableTo datetime NULL
AS
BEGIN
 IF (NOT EXISTS(SELECT * FROM Dishes WHERE DishID=@DishID))
 BEGIN
   DECLARE @errorMsg1 nvarchar(1024) = 'Dish with ID' + @DishID + 'does
not exist';
   THROW 52000, @errorMsg1, 1;
 END;
  BEGIN TRY
   INSERT INTO MenuItems (DishID, UnitPrice, UnitsOnOrder,
AvailableFrom, AvailableTo) VALUES (@DishID, @UnitPrice, @UnitsOnOrder,
@AvailableFrom, @AvailableTo);
 END TRY
 BEGIN CATCH
   DECLARE @errorMsg2 nvarchar(1024) = 'Error while inserting Menu Item:
'
   + ERROR_MESSAGE();
   THROW 52000, @errorMsg2, 1;
 END CATCH
END;
```

8. **DeleteFromCurrentMenu** - usuwa danie z menu

```
CREATE PROCEDURE DeleteItemFromCurrentMenu
 @ItemID int
AS
```

```
BEGIN
 IF NOT EXISTS(SELECT * FROM MenuItems WHERE ItemID = @ItemID)
   BEGIN
     DECLARE @errorMsg nvarchar(1024) = 'Dish does not exist in menu';
     THROW 52000, @errorMsg, 1;
   END;
 IF (SELECT AvailableTo FROM MenuItems WHERE ItemID = @ItemID) IS NOT
NULL
   BEGIN
     DECLARE @errorMsg1 nvarchar(1024) = 'Dish already not in current
menu';
     THROW 52000, @errorMsg1, 1;
   END;
 BEGIN TRY
   UPDATE MenuItems SET AvailableTo = GETDATE() WHERE ItemID = @ItemID
 END TRY
 BEGIN CATCH
   DECLARE @errorMsg2 nvarchar(1024) = 'Error while deleting dish from
current menu';
   THROW 52000, @errorMsg2, 1;
 END CATCH;
END;
```

9. **UpdateDiscountParam** - aktualizuje parametr zniżki

```
CREATE PROCEDURE UpdateDiscountParam
 @ParamName varchar(2),
 @Value int
AS
BEGIN
 IF NOT EXISTS(SELECT * FROM DiscountParamsDict WHERE
ParamName=@ParamName)
 BEGIN
   DECLARE @errorMsg1 nvarchar(1024) = 'Discount Param does not exist';
   THROW 52000, @errorMsg1, 1;
 END;
```

```
 DECLARE @ParamID int;

 SELECT @ParamID=(SELECT ParamID FROM DiscountParamsDict WHERE
ParamName=@ParamName);

 BEGIN TRY

   UPDATE DiscountParams SET AvailableTo=GETDATE() WHERE (AvailableTo IS
NULL) AND ParamID=@ParamID;

   INSERT INTO DiscountParams(ParamID, Value) VALUES (@ParamID, @Value);

 END TRY

 BEGIN CATCH

   DECLARE @ConstID int;

   SELECT @ConstID = (SELECT TOP 1 ConstID FROM DiscountParams WHERE
ParamID=@ParamID ORDER BY AvailableFrom DESC);

   UPDATE DiscountParams SET AvailableTo=NULL WHERE ConstID=@ConstID;

   DECLARE @errorMsg2 nvarchar(1024) = 'Error while inserting Discount
Param: '

   + ERROR_MESSAGE();

   THROW 52000, @errorMsg2, 1;

 END CATCH

END;
```

10. **GrantOneTimeDiscount** – przyznaje klientowi jednorazową zniżkę

```
CREATE PROCEDURE GrantOneTimeDiscount

 @CustomerID int

AS

BEGIN

 -- Check if the specified Customer is allowed to be granted a discount

 IF dbo.canCustomerGetOneTimeDiscount(@CustomerID) = 1

 BEGIN

   BEGIN TRY

     -- Get discount parameters

     DECLARE @DiscountValue int = dbo.getDiscountParamValue('R2');

     DECLARE @DiscountPeriod int = dbo.getDiscountParamValue('D1')

     DECLARE @AvailableFrom datetime = GETDATE();

     DECLARE @AvailableTo datetime = DATEADD(DAY, @DiscountPeriod,
@AvailableFrom);
```

```
    INSERT INTO OneTimeDiscount(CustomerID, DiscountValue,
AvailableFrom, AvailableTo)

    VALUES (@CustomerID, @DiscountValue, @AvailableFrom, @AvailableTo);

  END TRY

  BEGIN CATCH

    DECLARE @errorMsg1 nvarchar(1024) = 'Error while inserting to
OneTimeDiscount: '

    + ERROR_MESSAGE();

    THROW 52000, @errorMsg1, 1;

  END CATCH

 END

 ELSE

 BEGIN

  DECLARE @errorMsg2 nvarchar(1024) = CONCAT('Customer ', @CustomerID,
' is not eligible for One Time Discount');

  THROW 52000, @errorMsg2, 1;

 END

END;
```

11. **GrantPermanentDiscount** - przyznaje klientowi permanentną zniżkę

```
CREATE PROCEDURE GrantPermanentDiscount

 @CustomerID int

AS

BEGIN

 -- Check if the specified Customer already has been granted a Permanent
Discount

 IF EXISTS(SELECT * FROM PermanentDiscount WHERE CustomerID =
@CustomerID)

 BEGIN

  DECLARE @errorMsg1 nvarchar(1024) = CONCAT('Customer ', @CustomerID,
' has already been granted a Permanent Discount');

  THROW 52000, @errorMsg1, 1;

 END

 -- Check if the specified Customer is allowed to be granted a discount

 IF dbo.canCustomerGetPermanentDiscount(@CustomerID) = 1
```

```
 BEGIN
  BEGIN TRY
     -- Get discount parameters
     DECLARE @DiscountValue int = dbo.getDiscountParamValue('R1');
     DECLARE @AvailableFrom datetime = GETDATE();

     INSERT INTO PermanentDiscount(CustomerID, DiscountValue,
AvailableFrom)
     VALUES (@CustomerID, @DiscountValue, @AvailableFrom);
   END TRY
   BEGIN CATCH
     DECLARE @errorMsg2 nvarchar(1024) = 'Error while inserting to
PermanentDiscount: '
     + ERROR_MESSAGE();
     THROW 52000, @errorMsg2, 1;
   END CATCH;
 END;
 ELSE
 BEGIN
   DECLARE @errorMsg3 nvarchar(1024) = CONCAT('Customer ', @CustomerID,
' is not eligible for Permanent Discount');
   THROW 52000, @errorMsg3, 1;
 END
END;
```

12. **PlaceOrder** - dodaje nowe zamówienie

```
CREATE PROCEDURE PlaceOrder
 @CustomerID int,
 @EmployeeID int,
 @PaymentTypeID int,
 @IsTakeout bit,
 @OutDate datetime
AS
BEGIN
```

```
 IF NOT EXISTS(SELECT * FROM RestaurantEmployees WHERE
EmployeeID=@EmployeeID)

 BEGIN

   DECLARE @errorMsg1 nvarchar(1024) = 'Employee does not exist';

   THROW 52000, @errorMsg1, 1;

 END;

 IF @CustomerID IS NOT NULL

 BEGIN

   IF NOT EXISTS(SELECT * FROM Customers WHERE CustomerID=@CustomerID)

   BEGIN

     DECLARE @errorMsg2 nvarchar(1024) = 'Customer does not exist';

     THROW 52000, @errorMsg2, 1;

   END;

 END;

 IF @PaymentTypeID IS NOT NULL

 BEGIN

   IF NOT EXISTS(SELECT * FROM Payment WHERE
PaymentTypeID=@PaymentTypeID)

   BEGIN

     DECLARE @errorMsg3 nvarchar(1024) = 'Payment Method does not
exist';

     THROW 52000, @errorMsg3, 1;

   END;

 END;

 BEGIN TRY

   INSERT INTO Orders(CustomerID, EmployeeID, PaymentTypeID, OutDate)
VALUES (@CustomerID, @EmployeeID, @PaymentTypeID, @OutDate)

   DECLARE @OrderID int;

   SELECT @OrderID = SCOPE_IDENTITY();

   IF @IsTakeout=1

   BEGIN

     INSERT INTO TakeoutOrders(OrderID) VALUES (@OrderID);

   END;

 END TRY

 BEGIN CATCH
```

```
    DECLARE @errorMsg4 nvarchar(1024) = 'Error while adding Order: '
    + ERROR_MESSAGE();
    THROW 52000, @errorMsg4, 1;
 END CATCH
END;
```

13. **AddItemToOrder** - dodaje danie z menu do zamówienia

```
CREATE PROCEDURE AddItemToOrder
 @OrderID int,
 @ItemID int,
 @Quantity int
AS
BEGIN
 IF NOT EXISTS(SELECT * FROM CurrentMenuView WHERE ItemID=@ItemID)
   BEGIN
     DECLARE @errorMsg1 nvarchar(1024) = 'Menu Item does not exist';
     THROW 52000, @errorMsg1, 1;
   END;
 IF NOT EXISTS(SELECT * FROM Orders WHERE OrderID=@OrderID)
   BEGIN
     DECLARE @errorMsg2 nvarchar(1024) = 'Order does not exist';
     THROW 52000, @errorMsg2, 1;
   END;
 IF (SELECT CategoryName
     FROM MenuItems as mi
     INNER JOIN Dishes as d ON mi.DishID=d.DishID
     INNER JOIN Categories as c ON c.CategoryID=d.CategoryID
     WHERE mi.ItemID=@ItemID)='Seafood'
   BEGIN
     DECLARE @OutDate datetime;
     SELECT @OutDate = (SELECT OutDate FROM Orders WHERE
OrderID=@OrderID);
     DECLARE @MondayDate datetime;
```

```
      SELECT @MondayDate = DATEADD(DAY, -DATEPART(WEEKDAY, @OutDate)+2,
@OutDate);

      IF DATEPART(WEEKDAY,@OutDate) <5 OR DATEPART(WEEKDAY,@OutDate) >7

      BEGIN

        DECLARE @errorMsg3 nvarchar(1024) = 'Seafood can only be ordered
for a day between Thursday and Saturday';

        THROW 52000, @errorMsg3, 1;

      END;

      IF (SELECT OrderDate FROM Orders WHERE
OrderID=@OrderID)>@MondayDate

      BEGIN

        DECLARE @errorMsg4 nvarchar(1024) = 'Seafood can only be ordered
before Monday preceding order date';

        THROW 52000, @errorMsg4, 1;

      END;

   END;

   BEGIN TRY

      INSERT INTO OrderDetails(OrderID,ItemID,Quantity) VALUES
(@OrderID,@ItemID,@Quantity)

   END TRY

   BEGIN CATCH

      DECLARE @errorMsg nvarchar(1024) = 'Error while adding order item:
'

      + ERROR_MESSAGE();

      THROW 52000, @errorMsg, 1;

   END CATCH

END;
```

14. **PayOrder** - potwierdza opłacenie zamówienia (dodaje do zamówienia PaymentTypeID)

```
CREATE PROCEDURE PayOrder

 @OrderID int,

 @PaymentTypeID int

AS

BEGIN

 IF NOT EXISTS(SELECT * FROM Payment WHERE PaymentTypeID=@PaymentTypeID)
```

```
 BEGIN
   DECLARE @errorMsg1 nvarchar(1024) = 'Payment Type does not exist';
   THROW 52000, @errorMsg1, 1;
 END
 IF NOT EXISTS(SELECT * FROM Orders WHERE OrderID=@OrderID)
 BEGIN
   DECLARE @errorMsg2 nvarchar(1024) = 'Order does not exist';
   THROW 52000, @errorMsg2, 1;
 END
 IF (SELECT PaymentTypeID FROM Orders WHERE OrderID=@OrderID) IS NOT
NULL
 BEGIN
   DECLARE @errorMsg3 nvarchar(1024) = 'Order is already paid';
   THROW 52000, @errorMsg3, 1;
 END
 BEGIN TRY
   UPDATE Orders SET PaymentTypeID=@PaymentTypeID WHERE OrderID=@OrderID
 END TRY
 BEGIN CATCH
   DECLARE @errorMsg nvarchar(1024) = 'Error while setting payment type:
'
   + ERROR_MESSAGE();
   THROW 52000, @errorMsg, 1;
 END CATCH
END;
```

15. **ReceiveOrder** - dodaje do zamówienia datę jego wydania

```
CREATE PROCEDURE ReceiveOrder
 @OrderID int,
 @OutDate datetime
AS
BEGIN
 IF NOT EXISTS (SELECT OrderID FROM Orders WHERE OrderID = @OrderID)
   BEGIN
```

```
    DECLARE @errorMsg nvarchar(1024) = 'Order does not exist' +
ERROR_MESSAGE();

    THROW 52000, @errorMsg, 1;

  END;

 IF (SELECT OutDate FROM Orders WHERE OrderID = @OrderID) IS NOT NULL

  BEGIN

    DECLARE @errorMsg1 nvarchar(1024) = 'Order already receieved' +
ERROR_MESSAGE();

    THROW 52000, @errorMsg1, 1;

  END

  BEGIN TRY

    UPDATE Orders SET OutDate=@OutDate WHERE OrderID=@OrderID;

  END TRY

  BEGIN CATCH

    DECLARE @errorMsg2 nvarchar(1024) = 'Error while receiving order: '
+ ERROR_MESSAGE();

    THROW 52000, @errorMsg2, 1;

  END CATCH;

END;
```

16. **UpdateReservationConditions** - zmienia warunki rezerwacji

```
CREATE PROCEDURE UpdateReservationConditions

@MinValue int,

@MinOrdersNum int,

@MinPeopleNum int

AS

BEGIN

 BEGIN TRY

  UPDATE ReservationConditions SET MinValue=@MinOrdersNum,
MinOrdersNum=@MinOrdersNum, MinPeopleNum=@MinPeopleNum

 END TRY

 BEGIN CATCH

  DECLARE @errorMsg nvarchar(1024) = 'Error while updating reservation
conditions: '

  + ERROR_MESSAGE();
```

```
    THROW 52000, @errorMsg, 1;
 END CATCH
END;
```

17. **AddReservation** - dodaje nową rezerwację

```
CREATE PROCEDURE AddReservation
@CustomerID int,
@PeopleCount int,
@FromDate datetime
AS
BEGIN
 IF NOT EXISTS(SELECT * FROM Customers WHERE CustomerID=@CustomerID)
   BEGIN
     DECLARE @errorMsg1 nvarchar(1024) = 'Customer does not exist';
     THROW 52000, @errorMsg1, 1;
   END;
 IF @CustomerID in (SELECT CustomerID FROM IndividualCustomers)
 BEGIN
   IF (SELECT COUNT(*) FROM Orders WHERE CustomerID=@CustomerID)<(SELECT
MinOrdersNum FROM ReservationConditions)
     OR @PeopleCount<(SELECT MinPeopleNum FROM ReservationConditions)
   BEGIN
     DECLARE @errorMsg2 nvarchar(1024) = 'Customer does not meet the
requirements';
     THROW 52000, @errorMsg2, 2;
   END;
   IF NOT EXISTS (SELECT * FROM Orders WHERE CustomerID=@CustomerID AND
DATEDIFF(DAY,OutDate,@FromDate)=0)
   BEGIN
     DECLARE @errorMsg3 nvarchar(1024) = 'Individual Customer must place
an order with the reservation';
     THROW 52000, @errorMsg3, 2;
   END;
   IF (SELECT TotalPrice FROM IndividualCustomersOrdersView WHERE
CustomerID=@CustomerID AND  DATEDIFF(DAY,OutDate,@FromDate)=0)
```

```
   <(SELECT MinValue FROM ReservationConditions)

   BEGIN

     DECLARE @errorMsg4 nvarchar(1024) = 'Order does not meet the
requirements';

     THROW 52000, @errorMsg4, 2;

   END;

 END

 BEGIN TRY

   INSERT INTO
Reservations(CustomerID,PeopleCount,ReservationDate,FromDate) VALUES
(@CustomerID,@PeopleCount,GETDATE(),@FromDate)

 END TRY

   BEGIN CATCH

   DECLARE @errorMsg nvarchar(1024) = 'Error while adding reservation: '

   + ERROR_MESSAGE();

   THROW 52000, @errorMsg, 1;

 END CATCH

END;
```

18. **ConfirmIndividualReservation** - zatwierdza rezerwację dla klienta indywidualnego i przypisuje stolik

```
CREATE PROCEDURE ConfirmIndividualReservation

@ReservationID int,

@TableID int

AS

BEGIN

 IF NOT EXISTS(SELECT * FROM Tables WHERE TableID=@TableID)

   BEGIN

     DECLARE @errorMsg1 nvarchar(1024) = 'Table does not exist';

     THROW 52000, @errorMsg1, 1;

   END;

 IF NOT EXISTS(SELECT * FROM Reservations WHERE
ReservationID=@ReservationID)

   BEGIN

     DECLARE @errorMsg2 nvarchar(1024) = 'Reservation does not exist';
```

```
    THROW 52000, @errorMsg2, 1;

  END;

 IF (SELECT CustomerID FROM Reservations WHERE
ReservationID=@ReservationID) NOT IN (SELECT CustomerID FROM
IndividualCustomers)

 BEGIN

    DECLARE @errorMsg3 nvarchar(1024) = 'Not Individual Customer';

    THROW 52000, @errorMsg3, 1;

 END;

 IF (SELECT SeatsNum FROM Tables WHERE TableID = @TableID) != (SELECT
PeopleCount FROM Reservations WHERE ReservationID = @ReservationID)

  BEGIN

    DECLARE @errorMsg4 nvarchar(1024) = CONCAT('Table ', @TableID, '
does not have enough seats');

    THROW 52000, @errorMsg4, 1;

  END;

 DECLARE @FromDate datetime= (SELECT FromDate FROM Reservations where
ReservationID=@ReservationID )

 IF EXISTS(SELECT * FROM ReservedTablesView WHERE TableID=@TableID AND
DATEDIFF(DAY,FromDate,@FromDate)=0)

  BEGIN

    DECLARE @errorMsg5 nvarchar(1024) = CONCAT('Table ', @TableID, ' is
already reserved this day');

    THROW 52000, @errorMsg5, 1;

  end;

 BEGIN TRY

  INSERT INTO ReservationIndividuals(ReservationID,TableID) VALUES
(@ReservationID,@TableID)

  UPDATE Reservations SET ConfirmationDate=GETDATE() WHERE
ReservationID=@ReservationID

 END TRY

 BEGIN CATCH

  DELETE FROM ReservationIndividuals WHERE ReservationID=@ReservationID

  UPDATE Reservations SET ConfirmationDate=NULL WHERE
ReservationID=@ReservationID

  DECLARE @errorMsg nvarchar(1024) = 'Error while confirming
reservation: '

  + ERROR_MESSAGE();
```

```
    THROW 52000, @errorMsg, 1;
 END CATCH
END;
```

19. **AddCompanyReservationEmployee** -dodaje do rezerwacji firmowej pracownika firmy jako klienta

```
CREATE PROCEDURE AddCompanyReservationEmployee
@ReservationID int,
@GroupID int,
@CustomerID int
AS
BEGIN
 IF NOT EXISTS(
   SELECT *
   FROM Reservations
   WHERE ReservationID = @ReservationID
 )
   BEGIN
     DECLARE @errorMsg1 nvarchar(1024) = 'Reservation does not exist';
     THROW 52000, @errorMsg1, 1;
   END;

 IF NOT EXISTS(
   SELECT *
   FROM CompanyEmployees
   WHERE CustomerID = @CustomerID
 )
   BEGIN
     DECLARE @errorMsg2 nvarchar(1024) = CONCAT('Customer ',
@CustomerID, ' is not employed in the company of ID ', (
       SELECT CustomerID FROM Reservations WHERE ReservationID =
@ReservationID
     ));
     THROW 52000, @errorMsg2, 1;
```

```
    END;


  DECLARE @AddedPeopleCount int = (
    SELECT COUNT(CustomerID)
    FROM ReservationGroups
    WHERE ReservationID = @ReservationID
  );
  DECLARE @ExpectedPeopleCount int = (
    SELECT PeopleCount
    FROM Reservations
    WHERE ReservationID = @ReservationID
  );
  IF @AddedPeopleCount = @ExpectedPeopleCount
    BEGIN
      DECLARE @errorMsg3 nvarchar(1024) = 'Cannot add more people to the
reservation ' + STR(@ReservationID);
      THROW 52000, @errorMsg3, 1;
    END;


  IF EXISTS(
    SELECT *
    FROM ReservationGroups
    WHERE ReservationID = @ReservationID
      AND CustomerID = @CustomerID
  )
    BEGIN
      DECLARE @errorMsg4 nvarchar(1024) = CONCAT('Employee ',
@CustomerID, ' is already added to another group');
      THROW 52000, @errorMsg4, 1;
    END;


  BEGIN TRY
    INSERT INTO ReservationGroups(ReservationID,GroupID,CustomerID)
VALUES (@ReservationID,@GroupID,@CustomerID)
  END TRY
```

```
 BEGIN CATCH
   DECLARE @errorMsg nvarchar(1024) = 'Error while adding employee to
reservation: '
   + ERROR_MESSAGE();
   THROW 52000, @errorMsg, 1;
 END CATCH
END;
```

20. **AssignTableToCompanyNamedGroup** - przypisuje stolik do grupy osób z imiennej rezerwacji firmowej

```
CREATE PROCEDURE AssignTableToCompanyNamedGroup
@ReservationID int,
@GroupID int,
@TableID int
AS
BEGIN
 IF NOT EXISTS(SELECT * FROM Reservations WHERE
ReservationID=@ReservationID)
   BEGIN
     DECLARE @errorMsg1 nvarchar(1024) = 'Reservation does not exist';
     THROW 52000, @errorMsg1, 1;
   END;

 IF NOT EXISTS(SELECT * FROM ReservationGroups WHERE
ReservationID=@ReservationID AND GroupID=@GroupID)
   BEGIN
     DECLARE @errorMsg2 nvarchar(1024) = 'Group does not exist';
     THROW 52000, @errorMsg2, 12;
   END;

 IF NOT EXISTS(SELECT * FROM Tables WHERE TableID=@TableID)
   BEGIN
     DECLARE @errorMsg3 nvarchar(1024) = 'Table does not exist';
     THROW 52000, @errorMsg3, 1;
   END;
```

```
 DECLARE @SeatsNum int = (
   SELECT SeatsNum
   FROM Tables
   WHERE TableID = @TableID
 );
 DECLARE @GroupPeopleCount int = (
   SELECT COUNT(*)
   FROM ReservationGroups
   WHERE GroupID = @GroupID
     AND ReservationID = @ReservationID
 );
 IF @SeatsNum < @GroupPeopleCount
   BEGIN
     DECLARE @errorMsg4 nvarchar(1024) = CONCAT('Table ', @TableID, '
does not have enough seats');
     THROW 52000, @errorMsg4, 1;
   END;

 IF EXISTS(
   SELECT *
   FROM ReservationCompanies
   WHERE ReservationID = @ReservationID
     AND GroupID = @GroupID
 )
   BEGIN
     DECLARE @errorMsg5 nvarchar(1024) = 'Group is already assigned to a
table';
     THROW 52000, @errorMsg5, 1;
   END;

 DECLARE @FromDate datetime = (
   SELECT FromDate
   FROM Reservations
```

```
    WHERE ReservationID = @ReservationID
  );
  IF EXISTS(
    SELECT *
    FROM ReservedTablesView
    WHERE TableID = @TableID
      AND DATEDIFF(DAY, FromDate, @FromDate) = 0
  )
    BEGIN
      DECLARE @errorMsg6 nvarchar(1024) = CONCAT('Table ', @TableID, ' is
already reserved this day');
      THROW 52000, @errorMsg6, 1;
    END;

  BEGIN TRY
    INSERT INTO ReservationCompanies(ReservationID, GroupID, TableID,
GroupPeopleCount) VALUES (@ReservationID, @GroupID, @TableID,
@GroupPeopleCount)
  END TRY
  BEGIN CATCH
    DECLARE @errorMsg nvarchar(1024) = 'Error while assigning a table to
the named group: '
    + ERROR_MESSAGE();
    THROW 52000, @errorMsg, 1;
  END CATCH;
END;
```

21. **AssignTableToCompanyUnnamedGroup** - tworzy grupę osób z rezerwacji firmowej bezimiennej, siedzących przy jednym stoliku i przypisuje im stolik.

```
CREATE PROCEDURE AssignTableToCompanyUnnamedGroup
@ReservationID int,
@GroupPeopleCount int,
@TableID int
AS
BEGIN
```

```
 IF NOT EXISTS(SELECT * FROM Reservations WHERE
ReservationID=@ReservationID)
   BEGIN
     DECLARE @errorMsg1 nvarchar(1024) = 'Reservation does not exist';
     THROW 52000, @errorMsg1, 1;
   END;

 IF NOT EXISTS(SELECT * FROM Tables WHERE TableID=@TableID)
   BEGIN
     DECLARE @errorMsg2 nvarchar(1024) = 'Table does not exist';
     THROW 52000, @errorMsg2, 1;
   END;

 DECLARE @SeatsNum int = (
   SELECT SeatsNum
   FROM Tables
   WHERE TableID = @TableID
 );
 IF @SeatsNum < @GroupPeopleCount
   BEGIN
     DECLARE @errorMsg3 nvarchar(1024) = CONCAT('Table ', @TableID, '
does not have enough seats');
     THROW 52000, @errorMsg3, 1;
   END;

 DECLARE @FromDate datetime = (
   SELECT FromDate
   FROM Reservations
   WHERE ReservationID = @ReservationID
 );
 IF EXISTS(
   SELECT *
   FROM ReservedTablesView
   WHERE TableID = @TableID
```

```
      AND DATEDIFF(DAY, FromDate, @FromDate) = 0
  )
    BEGIN
      DECLARE @errorMsg4 nvarchar(1024) = CONCAT('Table ', @TableID, ' is
already reserved this day');
      THROW 52000, @errorMsg4, 1;
    END;

  DECLARE @GroupID int = (
    SELECT COUNT(GroupID) + 1
    FROM ReservationCompanies
    WHERE ReservationID = @ReservationID
  );
  BEGIN TRY
    INSERT INTO ReservationCompanies(ReservationID, GroupID, TableID,
GroupPeopleCount) VALUES (@ReservationID, @GroupID, @TableID,
@GroupPeopleCount)
  END TRY
  BEGIN CATCH
    DECLARE @errorMsg nvarchar(1024) = 'Error while assigning a table to
the unnamed group: '
    + ERROR_MESSAGE();
    THROW 52000, @errorMsg, 1;
  END CATCH;
END;
```

7. Funkcje

1. **GenerateOrderInvoice** - zwraca fakturę na zamówienie dla firmy

```
CREATE FUNCTION GenerateOrderInvoice (@OrderID int)
RETURNS @Invoice TABLE (ParamName varchar(150), ParamValue varchar(50))
AS
BEGIN
 -- General information
 DECLARE @CompanyName varchar(30);
 DECLARE @NIP varchar(15);
 DECLARE @Phone varchar(15);
 DECLARE @Email varchar(40);
 DECLARE @isTakeout bit;
 DECLARE @OrderDate datetime;
 DECLARE @OutDate datetime;

 SELECT
   @CompanyName = co.CompanyName,
   @NIP = co.NIP,
   @Phone = cu.Phone,
   @Email = cu.Email,
   @isTakeout = (
     SELECT CAST(
       IIF(tk.OrderID IS NULL, 1, 0)
     AS BIT)
   ),
   @OrderDate = o.OrderDate,
   @OutDate = o.OutDate
 FROM Customers AS cu
 INNER JOIN Companies AS co
 ON cu.CustomerID = co.CustomerID
 INNER JOIN Orders AS o
 ON o.CustomerID = co.CustomerID
 LEFT OUTER JOIN TakeoutOrders AS tk
```

```
 ON tk.OrderID = o.OrderID

 WHERE o.OrderID = @OrderID;


 INSERT INTO @Invoice VALUES ('Company name:', CAST(@CompanyName AS
varchar(50)));

 INSERT INTO @Invoice VALUES ('NIP:', CAST(@NIP AS varchar(50)));

 INSERT INTO @Invoice VALUES ('E-mail:', CAST(@Email AS varchar(50)));

 INSERT INTO @Invoice VALUES ('Phone:', CAST(@Phone AS varchar(50)));

 INSERT INTO @Invoice VALUES ('Order date:', CAST(@OrderDate AS
varchar(50)));

 INSERT INTO @Invoice VALUES ('Out date:', CAST(@OutDate AS
varchar(50)));


 -- Detailed list of ordered dishes

 DECLARE @ItemID int;

 DECLARE @DishName varchar(40);

 DECLARE @UnitPrice money;

 DECLARE @Quantity int;

 DECLARE @TotalItemAmount money;


 DECLARE CUR CURSOR FOR SELECT ItemID, Quantity FROM OrderDetails WHERE
OrderID = @OrderID

 OPEN CUR

 FETCH NEXT FROM CUR INTO @ItemID, @Quantity

 WHILE @@FETCH_STATUS=0

 BEGIN

   SELECT

     @DishName = d.DishName,

     @UnitPrice = mi.UnitPrice,

     @TotalItemAmount = @Quantity * mi.UnitPrice

   FROM MenuItems AS mi

   INNER JOIN Dishes AS d

   ON d.DishID = mi.DishID

   WHERE mi.ItemID = @ItemID;
```

```
    INSERT @Invoice VALUES (CONCAT('Dish name: ', @DishName,
                                   ', Quantity: ', @Quantity,
                                   ', Unit price: ', @UnitPrice,
                                   ', Total dish price: '), @Quantity *
@UnitPrice);

    FETCH NEXT FROM CUR INTO @ItemID, @Quantity
  END
  CLOSE CUR
  DEALLOCATE CUR

  -- Order summary
  DECLARE @TotalAmount money = dbo.getOrderTotalAmount(@OrderID);
  INSERT INTO @Invoice VALUES ('Total order amount: ', @TotalAmount);
  RETURN;
END;
```

2. **GenerateMonthlyInvoice** - zwraca miesięczną fakturę dla firmy

```
CREATE FUNCTION GenerateMonthlyInvoice (@CompanyID int)
RETURNS @Invoice TABLE (OrderID varchar(40), OrderValue money)
AS
BEGIN
DECLARE @CompanyName varchar(30) = (
    SELECT CompanyName FROM Companies WHERE CustomerID = @CompanyID);
DECLARE @NIP varchar(30) = (
    SELECT NIP FROM Companies WHERE CustomerID = @CompanyID);
DECLARE @Phone varchar(30) = (
    SELECT Phone FROM Customers WHERE CustomerID = @CompanyID);
DECLARE @Email varchar(30) = (
    SELECT Email FROM Customers WHERE CustomerID = @CompanyID);
INSERT INTO @Invoice VALUES (CONCAT('Company name: ',@CompanyName),NULL)
INSERT INTO @Invoice VALUES (CONCAT('NIP: ',@NIP),NULL)
INSERT INTO @Invoice VALUES (CONCAT('Phone: ',@Phone),NULL)
```

```
INSERT INTO @Invoice VALUES (CONCAT('E-mail: ',@Email),NULL)

INSERT INTO @Invoice VALUES ('Orders:',NULL)


DECLARE @OrderID int

DECLARE @OrderValue money

DECLARE CUR CURSOR FOR SELECT OrderID,TotalPrice FROM
CompanyCustomersMonthlyOrderReportView WHERE CustomerID=@CompanyID

OPEN CUR

FETCH NEXT FROM CUR INTO @OrderID, @OrderValue

WHILE @@FETCH_STATUS=0

BEGIN

 INSERT @Invoice VALUES (@OrderID, @OrderValue)

 FETCH NEXT FROM CUR INTO @OrderID, @OrderValue

END

CLOSE CUR

DEALLOCATE CUR

DECLARE @TotalValue money

SELECT @TotalValue=(SELECT SUM(TotalPrice) FROM
CompanyCustomersMonthlyOrderReportView WHERE CustomerID=@CompanyID)

INSERT INTO @Invoice VALUES ('Total value:',@TotalValue)

RETURN

END
```

3. **GetIndividualTablesReservationsStatistics** - zwraca statystyki rezerwacji stolików przez klientów indywidualnych za wskazaną liczbę poprzednich dni. Jeżeli `@DaysNum` jest nullem, statystyki dotyczą całego okresu działalności restauracji.

```
CREATE FUNCTION GetIndividualTablesReservationsStatistics (@DaysNum int
NULL)

RETURNS TABLE

AS

RETURN (

 SELECT ri.TableID, COUNT(*) AS IndividualReservations

 FROM Reservations AS r

 INNER JOIN ReservationIndividuals AS ri
```

```
 ON r.ReservationID = ri.ReservationID

 WHERE (@DaysNum IS NULL

   OR DATEDIFF(DAY, r.ReservationDate, GETDATE()) <= @DaysNum)

 GROUP BY ri.TableID

);
```

4. **GetCompanyTablesReservationsStatistics** - zwraca statystyki rezerwacji stolików przez klientów firmowych za wskazaną liczbę poprzednich dni. Jeżeli `@DaysNum` jest nullem, statystyki dotyczą całego okresu działalności restauracji.

```
CREATE FUNCTION GetCompanyTablesReservationsStatistics (@DaysNum int
NULL)

RETURNS TABLE

AS

RETURN (

 SELECT rc.TableID, COUNT(*) as CompanyReservations

 FROM Reservations as r

 INNER JOIN ReservationCompanies as rc on
r.ReservationID=rc.ReservationID

 WHERE (@DaysNum IS NULL OR DATEDIFF(DAY, r.ReservationDate, GETDATE())
<= @DaysNum)

 GROUP BY rc.TableID

);
```

5. **GetCustomerDiscountsStatistics** - zwraca statystyki, dotyczące liczby użytych zniżek przez klienta indywidualnego, pogrupowanych według typu i wartości zniżki za wskazaną liczbę poprzednich dni. Jeżeli `@DaysNum` jest nullem, statystyki dotyczą wszystkich wykorzystanych przez klienta zniżek.

```
CREATE FUNCTION GetCustomerDiscountsStatistics (@CustomerID int,
@DaysNum int NULL)
RETURNS TABLE
AS
RETURN (
 SELECT DiscountType, DiscountValue, COUNT(*) AS NumOfUses
 FROM UsedDiscountsView
 WHERE CustomerID = @CustomerID AND
   (@DaysNum IS NULL OR DATEDIFF(DAY, OrderDate, GETDATE()) <= @DaysNum)
 GROUP BY DiscountType, DiscountValue
);
```

6. **GetMenuStatistics** - zwraca statystyki (sumarycznie sprzedaną liczbę porcji oraz sumarycznie zarobioną kwotę) dla pozycji z menu za wskazaną liczbę poprzednich dni. Jeżeli `@DaysNum` jest nullem, statystyki dotyczą całego okresu działalności restauracji.

```
CREATE FUNCTION GetMenuStatistics (@DaysNum int NULL)

RETURNS TABLE

AS

RETURN (

 SELECT mi.ItemID, d.DishName, COUNT(*) AS Quantity, SUM(od.Quantity *
mi.UnitPrice * (1 - odtv.DiscountValue / 100)) AS TotalAMount

 FROM MenuItems AS mi

 INNER JOIN Dishes AS d

 ON d.DishID = mi.DishID

 INNER JOIN OrderDetails AS od

 ON od.ItemID = d.DishID

 INNER JOIN OrdersDiscountsTableView AS odtv

 ON odtv.OrderID = od.OrderID

 WHERE (@DaysNum IS NULL OR DATEDIFF(DAY, odtv.OrderDate, GETDATE()) <=
@DaysNum)

 GROUP BY mi.ItemID, d.DishName, mi.AvailableTo

);
```

7. **GetOrdersStatistics** - zwraca statystyki zamówień wybranego klienta (kwoty oraz daty złożenia zamówień) za wskazaną liczbę poprzednich dni. Jeżeli `@DaysNum` jest nullem, statystyki dotyczą wszystkich zamówień klienta.

```
CREATE FUNCTION GetOrdersStatistics (@CustomerID int, @DaysNum int NULL)

RETURNS TABLE

AS

RETURN (

 SELECT

   o.OrderID,

   SUM(od.Quantity * mi.UnitPrice * (1 - odtv.DiscountValue / 100)) AS
TotalAmount,

   o.OrderDate

 FROM OrdersDiscountsTableView AS odtv

 INNER JOIN OrderDetails AS od
```

```
 ON od.OrderID = odtv.OrderID
 INNER JOIN MenuItems AS mi
 ON mi.ItemID = od.ItemID
 INNER JOIN Orders AS o
 ON o.OrderID = od.OrderID
 WHERE o.CustomerID = @CustomerID
   AND (@DaysNum IS NULL OR DATEDIFF(DAY, odtv.OrderDate, GETDATE()) <=
@DaysNum)
 GROUP BY o.OrderID, o.OrderDate
);
```

8. **GetDiscountParamValue** - zwraca wartość wskazanego parametru zniżki

```
CREATE FUNCTION GetDiscountParamValue (@ParamName varchar(2))
RETURNS int
AS
BEGIN
 RETURN (
   SELECT Value
   FROM DiscountParamsTableView
   WHERE ParamName = @ParamName
     AND (AvailableTo IS NULL OR AvailableTo >= GETDATE())
 )
END;
```

9. **GetOrderTotalAmount** - zwraca sumaryczną kwotę wskazanego zamówienia

```
CREATE FUNCTION GetOrderTotalAmount (@OrderID int)
RETURNS money
AS
BEGIN
 RETURN (
   SELECT SUM(od.Quantity * mi.UnitPrice * (1 - odtv.DiscountValue /
100)) AS TotalAmount
   FROM OrdersDiscountsTableView AS odtv
```

```
    INNER JOIN OrderDetails AS od
    ON odtv.OrderID = od.OrderID
    INNER JOIN MenuItems AS mi
    ON od.ItemID = mi.ItemID
    WHERE odtv.OrderID = @OrderID
  )
END;
```

10. **GetAmountSpentByCustomer** - zwraca łączną kwotę wydaną przez klienta, liczoną od wskazanej daty początkowej. Jeżeli data początkowa jest nullem, zwracana jest sumarycznie wydana kwota przez klienta od pierwszego złożonego zamówienia.

```
CREATE FUNCTION GetAmountSpentByCustomer (@CustomerID int, @StartDate
datetime NULL)
RETURNS money
AS
BEGIN
  RETURN (
    SELECT ISNULL(SUM(dbo.GetOrderTotalAmount(OrderID)), 0) AS
TotalAmount
    FROM Orders
    WHERE CustomerID = @CustomerID AND
      (DATEDIFF(DAY, ISNULL(@StartDate, GETDATE()), OrderDate) >= 0
      OR @StartDate IS NULL)
  )
END;
```

11. **GetLastOneTimeDiscountStartDate** - zwraca poprzednią datę rozpoczęcia zniżki jednorazowej

```
CREATE FUNCTION GetLastOneTimeDiscountStartDate(@CustomerID int)
RETURNS datetime
AS
BEGIN
  RETURN (
```

```
    SELECT MAX(AvailableFrom)
    FROM CustomerOneTimeDiscountsView
    WHERE CustomerID = @CustomerID
  )
END;
```

12. **CanCustomerGetOneTimeDiscount** - zwraca wartość binarną, wskazującą, czy klient może otrzymać zniżkę jednorazową

```
CREATE FUNCTION CanCustomerGetOneTimeDiscount(@CustomerID int)
RETURNS bit
AS
BEGIN
 DECLARE @RequiredTotalAmount int = dbo.GetDiscountParamValue('K2');
 DECLARE @LastOneTimeDiscountStartDate datetime =
dbo.GetLastOneTimeDiscountStartDate(@CustomerID)
 DECLARE @TotalAmountSpent money =
dbo.GetAmountSpentByCustomer(@CustomerID,
@LastOneTimeDiscountStartDate);
 IF @TotalAmountSpent >= @RequiredTotalAmount AND @CustomerID IN (
   SELECT CustomerID FROM IndividualCustomers
 )
 BEGIN
   RETURN 1;
 END
 RETURN 0;
END;
```

13. **CanCustomerGetPermanentDiscount** - zwraca wartość binarną, wskazującą, czy klient może otrzymać dożywotnią zniżkę

```
CREATE FUNCTION CanCustomerGetPermanentDiscount(@CustomerID int)
RETURNS bit
AS
BEGIN
 DECLARE @RequiredOrdersNumber int = dbo.GetDiscountParamValue('Z1');
```

```
 DECLARE @MinOrderAmount int = dbo.GetDiscountParamValue('K1');
 IF @CustomerID IN (
     SELECT CustomerID FROM IndividualCustomers
   ) AND (SELECT COUNT(OrderID)
     FROM Orders
     WHERE CustomerID = @CustomerID AND
     dbo.GetOrderTotalAmount(OrderID) >= @MinOrderAmount) >
@RequiredOrdersNumber
 BEGIN
   RETURN 1;
 END
 RETURN 0;
END;
```

14. **IsHalfMenuItemsOlderThanTwoWeeks** - zwraca wartość binarną, wskazującą, czy więcej niż połowa pozycji w menu jest obecna w menu przynajmniej przez 2 tygodnie

```
CREATE FUNCTION IsHalfMenuItemsOlderThanTwoWeeks()
RETURNS bit
AS
BEGIN
 DECLARE @MenuLen int = (
   SELECT COUNT(*)
   FROM CurrentMenuView
   WHERE DATEDIFF(DAY, AvailableFrom, GETDATE()) >= 14
     AND DishID NOT IN (SELECT DishID FROM SeafoodMenuView)
 );
 DECLARE @DishesOlderThanTwoWeeksCount int = (
   SELECT COUNT(*)
   FROM DishesOlderThanTwoWeeksView
 );
 IF @DishesOlderThanTwoWeeksCount > CAST(@MenuLen AS FLOAT) / 2
 BEGIN
   RETURN 1;
```

```
 END
 RETURN 0;
END;
```

15. **CheckMenuItemsReplacementStatus** - zwraca tekstową informację, wskazującą, czy konieczna jest wymiana pozycji w menu

```
CREATE FUNCTION CheckMenuItemsReplacementStatus()
RETURNS varchar(55)
AS
BEGIN
 IF dbo.IsHalfMenuItemsOlderThanTwoWeeks() = 1
   BEGIN
     RETURN 'More than half of the menu items is at least 2 weeks old'
   END
 RETURN 'It is not necessary to replace menu items now'
END;
```

8. Triggery

1. **AddEmployeeToConfirmedReservation** - zapobiega dodaniu pracownika do zatwierdzonej rezerwacji firmowej

```
CREATE TRIGGER AddEmployeeToConfirmedReservation
ON ReservationGroups
AFTER INSERT
AS
BEGIN
 IF EXISTS(
   SELECT *
   FROM ReservationCompanies
   WHERE ReservationID = (SELECT ReservationID FROM INSERTED)
     AND GroupID = (SELECT GroupID FROM INSERTED)
   )
   BEGIN
     RAISERROR('People cannot be added to confirmed reservation', 1, 1)
     ROLLBACK TRANSACTION
   END
END;
```

2. **MultipleReservationsInOneDay** - zapobiega złożeniu więcej niż jednej rezerwacji na jeden dzień przez klienta

```
CREATE TRIGGER MultipleReservationsInOneDay
ON Reservations
AFTER INSERT
AS
BEGIN
 IF EXISTS(
   SELECT *
   FROM Reservations
   WHERE CustomerID = (SELECT CustomerID FROM INSERTED)
     AND DATEDIFF(DAY, FromDate, (SELECT FromDate FROM INSERTED)) = 0
     AND ReservationID != (SELECT ReservationID FROM INSERTED)
   )
   BEGIN
     RAISERROR('Customer already has made a reservation on this date',
1, 1)
     ROLLBACK TRANSACTION
   END
END;
```

3. **UpdateReservationConfirmationDate** - aktualizuje datę zatwierdzenia rezerwacji firmowej, kiedy wszystkie grupy pracowników zostały przypisane do stolików

```
CREATE TRIGGER UpdateReservationConfirmationDate
ON ReservationCompanies
AFTER INSERT
AS
BEGIN
 DECLARE @ReservationID int = (
   SELECT ReservationID FROM INSERTED
 );
 DECLARE @AllGroupsCount int = (
   SELECT COUNT(DISTINCT GroupID)
   FROM ReservationGroups
   WHERE ReservationID = @ReservationID
 );
 DECLARE @AssignedTableGroupsCount int = (
   SELECT COUNT(DISTINCT GroupID)
   FROM ReservationCompanies
   WHERE ReservationID = @ReservationID
 );

 IF @AllGroupsCount = @AssignedTableGroupsCount
 BEGIN
   UPDATE Reservations SET ConfirmationDate = GETDATE() WHERE
ReservationID = @ReservationID
 END
END;
```

4. **GrantDiscount**- przy składaniu zamówienia sprawdza, czy klient może otrzymać zniżkę

```
CREATE TRIGGER GrantDiscount
ON Orders
AFTER INSERT
AS
BEGIN
 DECLARE @CustomerID int=(SELECT CustomerID FROM INSERTED);
 IF dbo.CanCustomerGetOneTimeDiscount(@CustomerID) = 1
 BEGIN
   EXEC GrantOneTimeDiscount @CustomerID = @CustomerID
 END;
 IF dbo.CanCustomerGetPermanentDiscount(@CustomerID) = 1
 BEGIN
```

```
    EXEC GrantPermanentDiscount @CustomerID = @CustomerID
  END;
END;
```

5. **CheckIfItemAvailable**- przy dodawaniu dania do zamówienia sprawdza, czy można zamówić taką ilość tego dania

```
CREATE TRIGGER CheckIfItemAvailable
ON OrderDetails
AFTER INSERT
AS
BEGIN
  DECLARE @ItemID int = (SELECT ItemID FROM INSERTED);
  DECLARE @UnitsOnOrder int = (
    SELECT UnitsOnOrder
    FROM MenuItems
    WHERE ItemID = @ItemID
  );
  DECLARE @UnitsSold int = (
    SELECT SUM(Quantity)
    FROM OrderDetails AS od
    INNER JOIN Orders AS o
    ON o.OrderID = od.OrderID
    WHERE od.ItemID = @ItemID
    AND DATEDIFF(DAY, o.OrderDate, GETDATE()) = 0
  );
  IF @UnitsSold > @UnitsOnOrder
  BEGIN
    DECLARE @ErrorMsg nvarchar(100) = CONCAT('Daily order limit was
exceeded for an item with ID: ', @ItemID);
    RAISERROR(@ErrorMsg, 1, 1);
    ROLLBACK TRANSACTION
  END;
END;
```

6. **DeleteOrder**- przy usuwaniu zamówienia usuwa też odpowiednie rekordy z tabel OrderDetails i TakeoutOrders

```
CREATE TRIGGER DeleteOrder
ON Orders
INSTEAD OF DELETE
AS
BEGIN
  DECLARE @OrderID int = (SELECT OrderID FROM DELETED)
```

```
 DELETE FROM OrderDetails WHERE OrderID = @OrderID
 DELETE FROM TakeoutOrders WHERE OrderID = @OrderID
 DELETE FROM Orders WHERE OrderID = @OrderID
END;
```

9. Indeksy

1. **Orders** - CustomerID

```
CREATE NONCLUSTERED INDEX OrdersCustomerIDIndex ON Orders(CustomerID);
```

2. **OneTimeDiscount** - CustomerID, AvailableFrom, AvailableTo

```
CREATE CLUSTERED INDEX OneTimeDiscountCustomerIDIndex ON
OneTimeDiscount(CustomerID);

CREATE NONCLUSTERED INDEX OneTimeDiscountIndex ON
OneTimeDiscount(AvailableFrom, AvailableTo);
```

3. **PermanentDiscount** - CustomerID, AvailableFrom

```
CREATE CLUSTERED INDEX PermanentDiscountCustomerIDIndex ON
PermanentDiscount(CustomerID);

CREATE NONCLUSTERED INDEX PermanentDiscountIndex ON
PermanentDiscount(AvailableFrom);
```

4. **Reservations** - CustomerID, FromDate

```
CREATE NONCLUSTERED INDEX ReservationsIndex ON Reservations(CustomerID,
FromDate);
```

5. **MenuItems** - DishID, AvailableFrom, AvailableTo

```
CREATE NONCLUSTERED INDEX MenuItemsIndex ON
MenuItems(DishID,AvailableFrom, AvailableTo);
```

6. **DiscountParamsDict** - ParamName

```
CREATE NONCLUSTERED INDEX DiscountParamsDictIndex ON
DiscountParamsDict(ParamName);
```

7. **DiscountParams** - ParamID, AvailableFrom, AvailableTo

```
CREATE CLUSTERED INDEX DiscountParamsParamIDIndex ON
DiscountParams(ParamID);

CREATE NONCLUSTERED INDEX DiscountParamsIndex ON
DiscountParams(AvailableFrom, AvailableTo);
```

8. **Categories** - CategoryName

```
CREATE NONCLUSTERED INDEX CategoriesIndex ON Categories(CategoryName);
```

9. **OrderDetails** - ItemID

```
CREATE NONCLUSTERED INDEX OrderDetailsIndex ON OrderDetails(ItemID);
```

10. **RestaurantEmployees** - PersonID

```
CREATE NONCLUSTERED INDEX RestaurantEmployeesIndex ON
RestaurantEmployees(PersonID);
```

11. **IndividualCustomers** - PersonID

```
CREATE NONCLUSTERED INDEX IndividualCustomersIndex ON
IndividualCustomers(PersonID);
```

12. **CompanyEmployees** - CompanyID, PersonID

```
CREATE NONCLUSTERED INDEX CompanyEmployeesIndex ON
CompanyEmployees(CompanyID, PersonID);
```

13. **ReservationIndividuals** - TableID

```
CREATE NONCLUSTERED INDEX ReservationIndividualsIndex ON
ReservationIndividuals(TableID);
```

10. Uprawnienia

1. **Administrator**
    a. Dostęp do wszystkich tabel, widoków, procedur i funkcji
2. **Menedżer restauracji**
    a. Dostęp do wszystkich tabel i widoków
    b. Dodawanie pracowników
    c. Dodawanie nowych dań i kategorii
    d. Dodawanie/usuwanie dań z aktualnego menu
    e. Generowanie raportów
3. **Pracownik restauracji**
    a. Dostęp do tabel i widoków zamówień, rezerwacji, klientów, menu, zniżek
    b. Zatwierdzanie rezerwacji
4. **Klient indywidualny**
    a. Dostęp do menu i parametrów zniżek
    b. Możliwość zarejestrowania się
    c. Składanie zamówień i rezerwacji
5. **Firma**
    a. Dostęp do tabel menu i parametrów zniżek
    b. Możliwość zarejestrowania firmy
    c. Możliwość rejestrowania swoich pracowników
    d. Generowanie faktury na zamówienie/miesięcznej
    e. Składanie zamówień
    f. Składanie rezerwacji i dodawanie do nich pracowników
6. **Funkcje systemowe**
    a. Przyznawanie zniżek klientom indywidualnym
    b. Sprawdzanie czy należy dokonać zmian w menu