Podstawy baz danych - projekt i implementacja systemu bazodanowego dla firmy świadczącej usługi gastronomiczne 2020/2021

Autorzy: Marcin Kroczek, Adam Niemiec

Opis

Opis problemu

Projekt dotyczy systemu wspomagania działalności firmy świadczącej usługi gastronomiczne dla klientów indywidualnych oraz firm. System winien być możliwy do równoległego użytkowania przez kilka tego typu firm.

Ogólne informacje

W ofercie jest żywność (np. ciastka, lunch, drobne przekąski) oraz napoje bezalkoholowe (np. kawa, koktajle, woda). Usługi świadczone są na miejscu oraz na wynos. Zamówienie na wynos może być zlecone na miejscu lub z wyprzedzeniem (z wykorzystaniem formularza WWW i wyboru preferowanej daty i godziny odbioru zamówienia). Firma dysponuje ograniczoną liczbą stolików, w tym miejsc siedzących. Istnieje możliwość wcześniejszej rezerwacji stolika dla co najmniej dwóch osób. Z uwagi na zmieniające się ograniczenia związane z COVID-19, w poszczególnych dniach może być dostępna ograniczona liczba miejsc (w odniesieniu do powierzchni lokalu), zmienna w czasie.
Klientami są osoby indywidualne oraz firmy, odbierające większe ilości posiłków w porze lunchu lub jako catering (bez dostawy). Dla firm istnieje możliwość wystawienia faktury dla danego zamówienia lub faktury zbiorczej raz na miesiąc.

Menu

Menu ustalane jest co najmniej dziennym wyprzedzeniem. W firmie panuje zasada, że co najmniej połowa pozycji menu zmieniana jest co najmniej raz na dwa tygodnie, przy czym pozycja zdjęta może powtórzyć się nie wcześniej niż za 1 miesiąc. Ponadto, w dniach czwartek-piątek-sobota istnieje możliwość wcześniejszego zamówienia dań zawierających owoce morza. Z uwagi na indywidualny import takie zamówienie winno być złożone co maksymalnie do poniedziałku poprzedzającego zamówienie. Istnieje możliwość, że pozycja w menu zostanie usunięta na skutek wyczerpania się półproduktów. Wcześniejsza rezerwacja zamówienia/stolika
Internetowy formularz umożliwia klientowi indywidualnemu rezerwację stolika, przy jednoczesnym złożeniu zamówienia, z opcją płatności przed lub po zamówieniu, przy minimalnej wartości zamówienia 50 zł, w przypadku klientów, którzy dokonali wcześniej co najmniej 5 zamówień i/lub mniej, ale w tym przypadku na kwotę co najmniej 200 zł. Informacja wraz z potwierdzeniem zamówienia oraz wskazaniem stolika. Wysyłana jest po akceptacji przez obsługę. Internetowy formularz umożliwia także rezerwację stolików dla firm, w dwóch opcjach: rezerwacji stolików na firmę i/lub rezerwację stolików dla konkretnych pracowników firmy (imiennie).

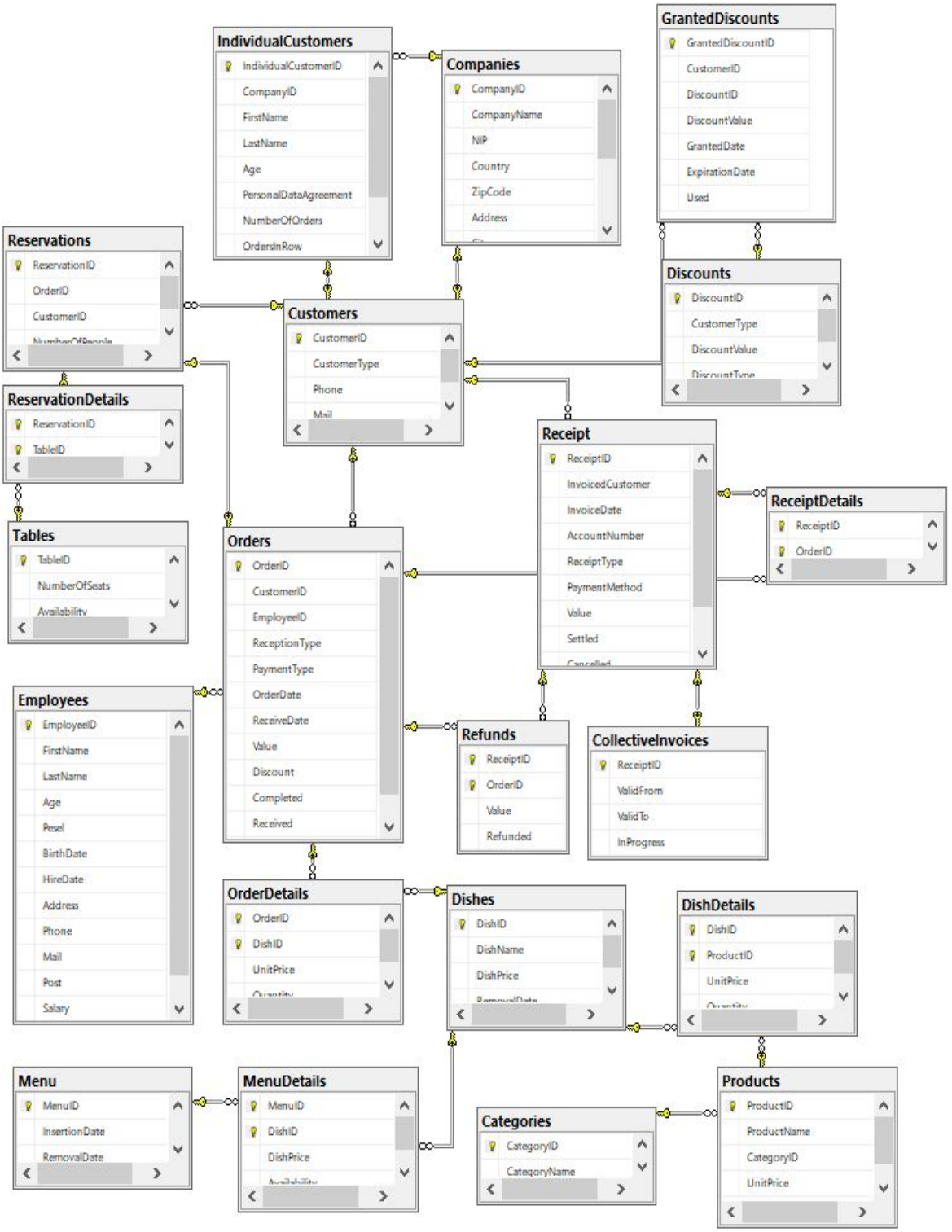
Rabaty
System umożliwia realizację programów rabatowych dla klientów indywidualnych:
Po realizacji ustalonej liczby zamówień Z1 (przykładowo Z1 = 10) za co najmniej określoną kwotę K1 (np. K1 = 30 zł każde zamówienie): R1% (np. R1 = 3%) zniżki na wszystkie zamówienia
Po realizacji kolejnych Z1 zamówień (np. Z1 = 10) za co najmniej określoną kwotę K1 każde (np. K1 = 30 zł): dodatkowe R1% zniżki na wszystkie zamówienia (np. R1 = 3%)
Po realizacji zamówień za łączną kwotę K2 (np. 1000 zł): jednorazowa zniżka R2% (np. 5%) na zamówienia złożone przez D1 dni (np. D1 = 7), począwszy od dnia przyznania zniżki
Po realizacji zamówień za łączną kwotę K3 (np. K3 = 5000 zł): jednorazowa zniżka R3% (np. R3 = 5%) na zamówienia złożone przez D2 dni (np. D2 = 7), począwszy od dnia przyznania zniżki.
W przypadku firm rabat udzielany jest zgodnie z zasadami:
Za każdy kolejny miesiąc, w którym dokonano co najmniej FZ zamówień (np. FZ = 5) za łączną kwotę co najmniej FK1 (np. FK1 = 500 zł): rabat FR1% (np. FR1 = 0,1%). W przypadku braku ciągłości w zamówieniach rabat zeruje się. Łączny, maksymalny rabat, to FM% (np. FM = 4%).
Za każdy kolejny kwartał, w którym dokonano zamówień za łączną kwotę FK2 (np. FK2 = 10000 zł): rabat kwotowy równy FR2% (np. FR2 = 5%) z łącznej kwoty, z którą zrealizowano zamówienie.

Raporty
System umożliwia generowanie raportów miesięcznych i tygodniowych, dotyczących rezerwacji stolików, rabatów, menu, a także statystyk zamówienia – dla klientów indywidualnych oraz firm – dotyczących kwot oraz czasu składania zamówień.
System umożliwia także generowanie raportów dotyczących zamówień oraz rabatów dla klienta indywidualnego oraz firm.

Uprawnienia
W celu optymalnego korzystania z bazy, przedstawiamy następujący podział na role i dostęp do poszczególnych danych:

1. Administrator systemu - posiada dostęp do całości systemu.
2. Pracownik działu obsługi klienta / działu gastronomicznego - posiada dostęp do danych związanych z zamówieniami, rezerwacjami, menu oraz produktami.
3. Pracownik księgowości - posiada dostęp do danych związanych z klientami, zamówieniami, rabatami, raportami oraz płatnościami.
4. Klient - posiada podstawowy dostęp do swoich danych, zamówień, rezerwacji oraz aktualnego menu firmy.

1. Schemat bazy:



2. Opisy tabel oraz warunki integralności:

Spis tabel:

1. Menu
2. MenuDetails
3. Dishes
4. DishDetails
5. Products
6. Categories
7. Orders
8. OrderDetails
9. Customers
10. IndividualCustomers
11. Companies
12. Employees
13. Discounts
14. GrantedDiscounts
15. Reservations
16. Tables
17. ReservationDetails
18. Receipt
19. ReceiptDetails
20. Refunds
21. CollectiveInvoices

1.Menu

Tabela z kluczem głównym **MenuID**, zawierająca daty dodania - **InsertionDate** oraz usunięcia - **RemovalDate** dla aktualnego oraz wszystkich poprzednich menu w restauracji.
Warunki integralnościowe:

- Data dodania przed datą usunięcia
- Data dodania większa o 2 tygodnie od daty usunięcia

```
CREATE TABLE Menu (
      MenuID int IDENTITY(1,1) NOT NULL,
      InsertionDate datetime NOT NULL DEFAULT DATEADD(DAY, 1,
GETDATE()),
      RemovalDate datetime NOT NULL,
CONSTRAINT [PK_Menu] PRIMARY KEY CLUSTERED
(
[MenuID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];

ALTER TABLE Menu
ADD CONSTRAINT CHK_Menu CHECK
(InsertionDate <= RemovalDate AND RemovalDate <= DATEADD(WEEK, 2,
InsertionDate));
```

2.MenuDetails

Tabela przypisująca każdemu menu - **MenuID** danie, jakie zawierało - **DishID**, tworząc w ten sposób klucz główny. Zawiera również cenę danego dania w momencie bycia w menu - **DishPrice** oraz informację o jego dostępności w przypadku aktualnego menu - **Availability**.
Warunki integralnościowe:

- Cena dania większa lub równa 0

```
CREATE TABLE MenuDetails (
```

```
      MenuID int NOT NULL,
      DishID int NOT NULL,
      DishPrice decimal(20,2) NOT NULL DEFAULT 0,
      Availability bit NOT NULL DEFAULT 1,
CONSTRAINT [PK_MenuDetails] PRIMARY KEY CLUSTERED (
[MenuID] ASC,
[DishID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY])
ON [PRIMARY] ;

ALTER TABLE MenuDetails
ADD FOREIGN KEY (MenuID) REFERENCES Menu(MenuID),
FOREIGN KEY (DishID) REFERENCES Dishes(DishID);

ALTER TABLE MenuDetails
ADD CONSTRAINT CHK_MenuDetails CHECK(DishPrice >= 0);
```

3.Dishes

Tabela zawierająca wszystkie dania z oferty restauracji. Posiada klucz główny **DishID**, przypisując każdemu daniu jego nazwę - **DishName**, aktualną cenę - **DishPrice** oraz datę jego usunięcia (jeżeli takie nastąpiło) - **RemovalDate**.
Warunki integralnościowe:

- Cena dania większa lub równa 0

```
CREATE TABLE Dishes (
      DishID int IDENTITY(1,1) NOT NULL,
      DishName nvarchar(40) NOT NULL UNIQUE,
      DishPrice decimal(20,2) NOT NULL,
      RemovalDate datetime DEFAULT NULL,
CONSTRAINT [PK_Dishes] PRIMARY KEY CLUSTERED
(
[DishID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];

ALTER TABLE Dishes
ADD CONSTRAINT CHK_Dishes CHECK(DishPrice >= 0);
```

4.DishDetails

Tabela przypisująca daniu - **DishID** wszystkie produkty w nim zawarte - **ProductID** tworząc podwójny klucz główny. Posiada również cenę produktu z momentu przypisania - **UnitPrice** oraz ilość jest sztuk w daniu - **Quantity**.
Warunki integralnościowe:

- Cena dania większa od 0
- Ilość każdego produktu w daniu większa lub równa 1

```
CREATE TABLE DishDetails (
      DishID int,
      ProductID int,
      UnitPrice decimal(20,2) NOT NULL,
      Quantity int DEFAULT 1,
CONSTRAINT [PK_DishDetails] PRIMARY KEY CLUSTERED (
[DishID] ASC,
[ProductID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY])
ON [PRIMARY] ;

ALTER TABLE DishDetails
ADD FOREIGN KEY (DishID) REFERENCES Dishes(DishID),
FOREIGN KEY (ProductID) REFERENCES Products(ProductID);

ALTER TABLE DishDetails
ADD CONSTRAINT CHK_DishDetails CHECK(UnitPrice > 0 AND Quantity >= 1);
```

5.Products

Tabela zawierająca produkty z kluczem głównym - **ProductID**. Zawiera nazwę produktu - **ProductName**, ID kategorii - CategoryID, cenę jednostkową - **UnitPrice** oraz aktualną ilość sztuk w magazynie - **UnitsInStock**.
Warunki integralnościowe:

- Cena produktu większa od 0
- Ilość produktu w magazynie większa lub równa 0

```
CREATE TABLE Products (
      ProductID int IDENTITY(1,1) NOT NULL,
      ProductName nvarchar(50) NOT NULL,
      CategoryID int NOT NULL,
      UnitPrice decimal(20,2) NOT NULL,
      UnitsInStock smallint DEFAULT 0,
CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED
(
[ProductID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];
```

```
ALTER TABLE Products
ADD FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID);

ALTER TABLE Products
ADD CONSTRAINT CHK_Products CHECK(UnitPrice > 0 AND UnitsInStock >= 0);
```

6.Categories

Tabela z kluczem głównym - **CategoryID**, przypisująca każdej kategorii jej nazwę - **CategoryName**.

```
CREATE TABLE Categories (
      CategoryID int IDENTITY(1,1) NOT NULL,
      CategoryName nvarchar(30) NOT NULL UNIQUE,
CONSTRAINT [PK_Categories] PRIMARY KEY CLUSTERED
(
[CategoryID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;
```

7.Orders

Tabela zamówień z kluczem głównym - **OrderID**, zawierająca informacje o złożonym zamówieniu. Opisują ją pola: **CustomerID** - ID klienta składającego zamówienie, **EmployeeID** - ID pracownika realizującego zamówienie, **ReceptionType** - sposób odbioru, **PaymentType** - sposób zapłaty, **OrderDate** - data złożenia zamówienia, **ReceiveDate** - data na którą złożono zamówienie, **Value** - wartość zamówienia, **Discount** - wartość zniżek przysługujących klientowi, **Completed** - informacja o zakończeniu składania zamówienia, **Received** - informacja o odebraniu zamówienia przez klienta, **Settled** - informacja o opłaceniu należności za zamówienie.
Warunki integralnościowe:

- Data złożenia zamówienia mniejsza od daty jego realizacji
- Wartość zamówienia większa lub równa 0
- Wartość zniżek większa lub równa 0
- Wartość zniżek mniejsza lub równa wartości zamówienia

```
CREATE TABLE Orders (
      OrderID int IDENTITY(1,1) NOT NULL,
      CustomerID int NOT NULL,
      EmployeeID int,
      ReceptionType nvarchar(11) NOT NULL
      CHECK (ReceptionType IN
      ('takeaway', 'delivery', 'reservation')) DEFAULT 'takeaway',
      PaymentType nvarchar(11) NOT NULL
      CHECK (PaymentType IN ('on-delivery', 'in-advance')),
```

```
	OrderDate datetime DEFAULT GETDATE(),
	ReceiveDate datetime NOT NULL,
	Value decimal(20,2) NOT NULL,
	Discount decimal(20,2),
	Completed bit NOT NULL DEFAULT 0,
	Received bit DEFAULT NULL,
	Settled bit NOT NULL DEFAULT 0,
CONSTRAINT [PK_Orders] PRIMARY KEY CLUSTERED
(
[OrderID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]) ON [PRIMARY]
;

ALTER TABLE Orders
ADD FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);

ALTER TABLE Orders
ADD CONSTRAINT CHK_Orders CHECK(OrderDate <= ReceiveDate
AND Value >= 0 AND Discount >= 0 AND Discount <= Value);
```

8.OrderDetails

Tabela z podwójnym kluczem głównym, przypisująca każdemu zamówieniu - **OrderID** dania wchodzące w jego skład - **DishID**, ich cenę jednostkową - **UnitPrice**, oraz ilość dań w zamówieniu - **Quantity**.

```
CREATE TABLE OrderDetails (
	OrderID int,
	DishID int,
	UnitPrice decimal(20,2) NOT NULL,
	Quantity smallint DEFAULT 1,
CONSTRAINT [PK_OrderDetails] PRIMARY KEY CLUSTERED (
[OrderID] ASC,
[DishID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY])
ON [PRIMARY] ;

ALTER TABLE OrderDetails
ADD FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
FOREIGN KEY (DishID) REFERENCES Dishes(DishID);
```

9.Customers

Tabela zawierająca wszystkich klientów restauracji tabel z klientami indywidualnymi oraz firmami z kluczem głównym **CustomerID**. Posiada podstawowe informacje o kliencie: **CustomerType** - typ klienta, **Phone** oraz **Mail**.
Warunki integralnościowe:

- Długość numeru telefonu wynosi 9
- Numer telefonu zawiera znaki od 0 do 9
- Email wygląda jak w szablonie ‘%@%.%’

```
CREATE TABLE Customers (
      CustomerID int IDENTITY(1,1) NOT NULL,
      CustomerType nvarchar (10) NOT NULL
      CHECK (CustomerType IN ('Individual', 'company')) DEFAULT
'individual',
      Phone nvarchar(9) NOT NULL DEFAULT 'forbidden',
      Mail nvarchar(30) NOT NULL DEFAULT 'forbidden',
CONSTRAINT [PK_Customers] PRIMARY KEY CLUSTERED
(
[CustomerID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];

ALTER TABLE Customers
ADD CONSTRAINT CHK_Customers CHECK(LEN(Phone) = 9 AND
Phone LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' AND Mail LIKE
'%@%.%');
```

10.IndividualCustomers

Tabela klientów indywidualnych z kluczem głównym **IndividualCustomerID**. Zawiera następujące informacje o klientach: **CompanyID** - ID firmy, jeśli klient w jakiejś pracuje, dane personalne - **FirstName**, **LastName**, **Age**, zgodę na przetwarzanie danych osobowych - **PersonalDataAgreement** oraz statystyki zamówień klienta, na których podstawie przyznawane są rabaty: **NumberOfOrders**, **OrdersInRow**, **SumOfOrders**.
Warunki integralnościowe:

- Wiek większy lub równy 12
- Liczba zamówień większa lub równa 0
- Liczba kolejnych zamówień większa lub równa 0
- Suma wartości zamówień większa lub równa 0

```
CREATE TABLE IndividualCustomers (
      IndividualCustomerID int IDENTITY(1,1) NOT NULL,
      CompanyID int DEFAULT NULL,
      FirstName nvarchar(24) NOT NULL DEFAULT 'forbidden',
```

```
	LastName nvarchar(30) NOT NULL DEFAULT 'forbidden',
	Age int DEFAULT NULL,
	PersonalDateAgreement bit NOT NULL DEFAULT 0,
	NumberOfOrders int NOT NULL DEFAULT 0,
	OrdersInRow int NOT NULL DEFAULT 0,
	SumOfOrders decimal(20,2) NOT NULL DEFAULT 0,
CONSTRAINT [PK_IndividualCustomers] PRIMARY KEY CLUSTERED
(
[IndividualCustomerID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];

ALTER TABLE IndividualCustomers
ADD FOREIGN KEY (CompanyID) REFERENCES Companies(CompanyID),
FOREIGN KEY (IndividualCustomerID) REFERENCES Customers(CustomerID);

ALTER TABLE IndividualCustomers
ADD CONSTRAINT CHK_IndividualCustomers CHECK(Age >= 12 AND
NumberOfOrders >= 0 AND OrdersInRow >= 0 AND SumOfOrders >= 0);
```

11.Companies

Tabela firm z kluczem głównym **CompanyID**. Zawiera następujące dane firm: **CompanyName** - nazwa firmy, **NIP**, **Country** - kraj, **ZipCode** - kod pocztowy, **Address** - adres, **City** - miasto, **InvoicePeriod** - okres rozliczania faktur zbiorczych oraz statystyki zamówień firmy, na których podstawie przyznawane są rabaty: **NumberOfOrdersInMonth**, **SumOfOrdersInMonth**, **NumberOfMonthsInRow**, **SumOfOrdersInQuarter**.
Warunki integralnościowe:

- Długość NIP równa 10
- NIP oraz kod pocztowy zawierą znaki od 0 do 9
- Okres rozliczania (faktury zbiorcze) wynosi -1 lub od 1 do 12
- Ilość zamówień w miesiącu większa lub równa 0
- Suma wartości zamówień w miesiącu większa lub równa 0
- Ilość kolejnych miesięcy(ciągłość rabatu nr5) większa lub równa 0
- Suma wartości zamówień w kwartale większa lub równa 0

```
CREATE TABLE Companies (
	CompanyID int IDENTITY(1,1) NOT NULL,
	CompanyName nvarchar(30) NOT NULL,
	NIP nvarchar(10) NOT NULL UNIQUE,
	Country nvarchar(40) NOT NULL,
	ZipCode nvarchar(30) NOT NULL UNIQUE,
	Address nvarchar(40) NOT NULL UNIQUE,
	City nvarchar(40) NOT NULL,
	InvoicePeriod int NOT NULL DEFAULT -1,
	NumberOfOrdersInMonth int NOT NULL DEFAULT 0,
```

```
      SumOfOrdersInMonth decimal(20,2) NOT NULL DEFAULT 0,
      NumberOfMonthsInRow int NOT NULL DEFAULT 0,
      SumOfOrdersInQuarter decimal(20,2) NOT NULL DEFAULT 0,
CONSTRAINT [PK_Companies] PRIMARY KEY CLUSTERED
(
[CompanyID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY],
CONSTRAINT [UniqueNIP_Companies] UNIQUE NONCLUSTERED (
[NIP] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY];

ALTER TABLE Companies
ADD FOREIGN KEY (CompanyID) REFERENCES Customers(CustomerID);

ALTER TABLE Companies
ADD CONSTRAINT CHK_Companies CHECK(LEN(NIP) = 10 AND
NIP LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' AND
ZipCode LIKE '[0-9][0-9]-[0-9][0-9][0-9]' AND InvoicePeriod >= -1 AND
InvoicePeriod != 0 AND InvoicePeriod <= 12 AND NumberOfOrdersInMonth >=
0 AND
SumOfOrdersInMonth >= 0 AND NumberOfMonthsInRow >= 0 AND
SumOfOrdersInQuarter >= 0 );
```

12.Employees

Tabela pracowników z kluczem głównym - EmployeeID. Posiada następujące informacje o pracownikach: dane personalne - **FirstName**, **LastName**, **Pesel**, **BirthDate**, **HireDate**, **Address**, **Phone**, **Mail**, posadę pracownika - **Post** oraz jego wynagrodzenie - **Salary**. Warunki integralnościowe:

- Różnica daty zatrudnienia i daty urodzin większa lub równa 16
- Wynagrodzenie większe lub równe 2600
- Długość numeru telefonu równa 9
- Długość numeru pesel równa 11
- Telefon pesel mają znaki od 0 do 9
- Email wygląda jak w szablonie ‘%@%.%’

```
CREATE TABLE Employees (
      EmployeeID int IDENTITY(1,1) NOT NULL,
      FirstName nvarchar(30) NOT NULL,
      LastName nvarchar(30) NOT NULL,
      Pesel nvarchar(11) NOT NULL UNIQUE,
      BirthDate datetime NOT NULL,
```

```
      HireDate datetime NOT NULL,
      Address nvarchar(60) NOT NULL,
      Phone nvarchar(9) NOT NULL UNIQUE,
      Mail nvarchar(30) NOT NULL UNIQUE,
      Post nvarchar(20) NOT NULL
      CHECK (Post IN
      ('manager', 'cook', 'customer service', 'waiter', 'delivery',
'other')) DEFAULT 'other',
      Salary decimal(20,2) NOT NULL DEFAULT 2600,
CONSTRAINT [PK_Employees] PRIMARY KEY CLUSTERED (
[EmployeeID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY],
CONSTRAINT [UniquePhone_Employees] UNIQUE NONCLUSTERED (
[Phone] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY],
CONSTRAINT [UniqueMail_Employees] UNIQUE NONCLUSTERED (
[Mail] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY],
CONSTRAINT [UniquePesel_Employees] UNIQUE NONCLUSTERED (
[Pesel] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
)ON [PRIMARY] ;

ALTER TABLE Employees
ADD CONSTRAINT CHK_Employees CHECK((YEAR(HireDate)- YEAR(BirthDate)) >=
16 AND Salary >= 2600
AND LEN(Phone) = 9 AND LEN(PESEL) = 11 AND
Pesel LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' AND
Phone LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' AND Mail LIKE
'%@%.%');
```

13.Discounts

Tabela rabatów przyznawanych przez restaurację z kluczem głównym **DiscountID**. Posiada informacje o typie klienta, dla której zniżka jest przyznana - **CustomerType**, wartość zniżki - **DiscountValue** oraz jej typ (procentowy, kwotowy) - **DiscountType**.

```
CREATE TABLE Discounts (
      DiscountID int IDENTITY(1,1) NOT NULL,
      CustomerType nvarchar (10) NOT NULL
```

```
      CHECK (CustomerType IN ('individual', 'company')) DEFAULT
'individual',
      DiscountValue decimal(3, 2) NOT NULL,
      DiscountType nvarchar(11) NOT NULL
      CHECK (DiscountType IN ('percentage', 'amount')),
CONSTRAINT [PK_Discounts] PRIMARY KEY CLUSTERED
(
[DiscountID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;
```

14.GrantedDiscounts

Tabela wszystkich przyznanych przez restaurację zniżek z kluczem głównym **GrantedDiscountID**. Przechowuje informacje o kliencie, któremu został przyznany rabat - **CustomerID**, numerze rabatu - **DiscountID**, wartości przyznanej zniżki - **DiscountValue**, datach przyznania - **GrantedDate** oraz przedawnienia - **ExpirationDate** oraz informację, czy rabat został zużyty.
Warunki integralnościowe:

- Data przyznania zniżki mniejsza od daty jej wygnaśnięcia

```
CREATE TABLE GrantedDiscounts (
      GrantedDiscountID int IDENTITY(1,1) NOT NULL,
      CustomerID int NOT NULL,
      DiscountID int NOT NULL,
      DiscountValue decimal(3, 2) NOT NULL,
      GrantedDate datetime NOT NULL DEFAULT GETDATE(),
      ExpirationDate datetime DEFAULT NULL,
      Used bit NOT NULL DEFAULT 0,
CONSTRAINT [PK_GrantedDiscounts] PRIMARY KEY CLUSTERED
(
[GrantedDiscountID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;

ALTER TABLE GrantedDiscounts
ADD FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),
FOREIGN KEY (DiscountID) REFERENCES Discounts(DiscountID);

ALTER TABLE GrantedDiscounts
ADD CONSTRAINT CHK_GrantedDiscounts CHECK(GrantedDate < ExpirationDate);
```

15.Reservations

Tabela rezerwacji z kluczem głównym **ReservationID**. Zawiera numer zamówienia przypisany do rezerwacji - **OrderID**, numer klienta rezerwującego stolik - **CustomerID** oraz liczbę gości przypisanych do rezerwacji - **NumberOfPeople**.

```
CREATE TABLE Reservations (
      ReservationID int IDENTITY(1,1) NOT NULL,
      OrderID int NOT NULL,
      CustomerID int NOT NULL,
      NumberOfPeople int NOT NULL,
CONSTRAINT [PK_Reservations] PRIMARY KEY CLUSTERED
(
[ReservationID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;

ALTER TABLE Reservations
ADD FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);

ALTER TABLE Reservations
ADD CONSTRAINT CHK_Reservations CHECK(NumberOfPeople >=1);

CREATE TABLE Tables (
      TableID int IDENTITY(1,1) NOT NULL,
      NumberOfSeats int NOT NULL,
      Availability bit NOT NULL DEFAULT 0,
CONSTRAINT [PK_Tables] PRIMARY KEY CLUSTERED
(
[TableID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;
```

16.Tables

Tabela stolików z kluczem głównym **TableID**. Posiada informacje o ilości wolnych miejsc - **NumberOfSeats** i aktualnej dostępności stolika - **Availability**.
Warunki integralnościowe:

- Ilość miejsc dla stolika mieści się w przedziale od 1 do 10

```
CREATE TABLE Tables (
      TableID int IDENTITY(1,1) NOT NULL,
      NumberOfSeats int NOT NULL,
      Availability bit NOT NULL DEFAULT 0,
CONSTRAINT [PK_Tables] PRIMARY KEY CLUSTERED
```

```
(
[TableID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;

ALTER TABLE Tables
ADD CONSTRAINT CHK_Tables CHECK(NumberOfSeats >= 1 AND NumberOfSeats <= 10);
```

17.ReservationDetails

Tabela z podwójnym kluczem głównym, przypisująca rezerwacjom - **ReservationID** stoliki - **TableID**.

```
CREATE TABLE ReservationDetails (
      ReservationID int NOT NULL,
      TableID int NOT NULL,
CONSTRAINT [PK_ReservationDetails] PRIMARY KEY CLUSTERED (
[ReservationID] ASC,
[TableID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY])
ON [PRIMARY] ;

ALTER TABLE ReservationDetails
ADD FOREIGN KEY (TableID) REFERENCES Tables(TableID),
FOREIGN KEY (ReservationID) REFERENCES Reservations(ReservationID);
```

18.Receipt

Tabela paragonów oraz faktur wystawianych klientom z kluczem głównym **ReceiptID**. Zawiera następujące informacje: **InvoicedCustomer** - ID klienta na którego wystawiono dokument, **InvoiceDate** - data wystawienia, **AccountNumber** - numer konta bankowego klienta, **ReceiptType** - rodzaj dokumentu (Paragon, faktura, faktura zbiorcza), **PaymentMethod** - metoda płatności, **Value** - należność, **Settled** - informacja o zapłacie należności, **Cancelled** - informacja o anulowaniu zamówienia, **SaleDate** - data sprzedaży. Warunki integralnościowe:

- Długość konta bankowego wynosi 26

```
CREATE TABLE Receipt (
      ReceiptID int IDENTITY(1,1) NOT NULL,
      InvoicedCustomer int NOT NULL,
      InvoiceDate datetime,
      AccountNumber nvarchar(26),
      ReceiptType nvarchar(20) NOT NULL
      CHECK (ReceiptType IN
      ('receipt', 'one-time invoice', 'collective invoice')),
      PaymentMethod nvarchar(8) NOT NULL
      CHECK (PaymentMethod IN ('cash', 'card', 'transfer')),
```

```
	Value decimal(20,2) NOT NULL,
	Settled bit NOT NULL DEFAULT 0,
	Cancelled bit NOT NULL DEFAULT 0,
	SaleDate datetime,
CONSTRAINT [PK_Receipt] PRIMARY KEY CLUSTERED
(
[ReceiptID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;

ALTER TABLE Receipt
ADD FOREIGN KEY (InvoicedCustomer) REFERENCES Customers(CustomerID);

ALTER TABLE Receipt
ADD CONSTRAINT CHK_Receipt CHECK( LEN(AccountNumber) = 26);
```

19.ReceiptDetails

Tabela z podwójnym kluczem głównym, przypisująca fakturom - **ReceiptID** zamówienia - **OrderID**.

```
CREATE TABLE ReceiptDetails (
	ReceiptID int NOT NULL,
	OrderID int NOT NULL,
CONSTRAINT [PK_ReceiptDetails] PRIMARY KEY CLUSTERED (
[ReceiptID] ASC,
[OrderID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY])
ON [PRIMARY] ;

ALTER TABLE ReceiptDetails
ADD FOREIGN KEY (ReceiptID) REFERENCES Receipt(ReceiptID),
FOREIGN KEY (OrderID) REFERENCES Orders(OrderID);
```

20.Refunds

Tabela z podwójnym kluczem głównym, przypisująca fakturom anulowanych zamówień - **ReceiptID** zamówienia - **OrderID**. Posiada również informację o należnej do zwrócenia wartości - **Value** oraz informacji o potwierdzeniu wykonania zwrotu - **Refunded**.

```
CREATE TABLE Refunds(
	ReceiptID int NOT NULL,
	OrderID int NOT NULL,
	Value decimal(20,2) NOT NULL DEFAULT 0,
	Refunded bit NOT NULL DEFAULT 0,
CONSTRAINT [PK_Refunds] PRIMARY KEY CLUSTERED (
[ReceiptID] ASC,
[OrderID] ASC
```

```
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY])
ON [PRIMARY] ;

ALTER TABLE Refunds
ADD FOREIGN KEY (ReceiptID) REFERENCES Receipt(ReceiptID),
FOREIGN KEY (OrderID) REFERENCES Orders(OrderID);
```

21.CollectiveInvoices

Table zawierająca faktury zbiorcze firm z kluczem głównym **ReceiptID**. Zawiera informacje o początku i końcu okresu, który dana faktura obejmuje - **ValidFrom**, **ValidTo** oraz informację, czy dany okres rozliczeniowy jeszcze trwa - **InProgress**.
Warunki integralnościowe:

- Data rozpoczynająca okres rozliczeniowy mniejsza od daty kończącej

```
CREATE TABLE CollectiveInvoices(
      ReceiptID int NOT NULL,
      ValidFrom datetime NOT NULL,
      ValidTo datetime NOT NULL,
      InProgress bit NOT NULL DEFAULT 1,
CONSTRAINT [PK_CIReceipt] PRIMARY KEY CLUSTERED
(
[ReceiptID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY] ;


ALTER TABLE CollectiveInvoices
ADD CONSTRAINT CHK_CollectiveInvoices CHECK(ValidTo > ValidFrom);

ALTER TABLE CollectiveInvoices
ADD FOREIGN KEY (ReceiptID) REFERENCES Receipt(ReceiptID);
```

3. Procedury:

Spis procedur:

1. AddCustomer
2. UpdateCustomerData
3. AddDiscount
4. AddGrantedDiscount
5. UpdateUsed
6. UpdateExpiredDiscounts
7. GetDiscountValue
8. AddIndividualCustomer
9. UpdateIndividualCustomerData
10. UpdateIndCustomerOrdersData
11. AddCompany
12. UpdateCompanyData
13. UpdateCompanyOrdersData
14. MonthlyCompanyUpdate
15. AddEmployee
16. UpdateEmployeeData
17. AddReservation
18. AddTableToReservation
19. AddTable
20. UpdateAvailability
21. UpdateDataForDiscounts
22. AddCategory
23. RemoveMenu
24. AddMenu
25. AddDishToMenu
26. UpdateDishAvailability
27. AddProduct
28. UpdateUnitsInStock
29. UpdateUnitPrice
30. AddDish
31. UpdateDishPrice
32. RemoveDish
33. AddProductToDish
34. ChangeReceiptValue
35. AddOrderToReceipt
36. RemoveOrderFromReceipt
37. SettleOrder
38. AddCollectiveInvoice
39. AddReceipt
40. SettleReceipt
41. InvoiceReceipt
42. AddDiscountToOrder
43. AddOrder
44. InvoiceOrder
45. CompleteOrder
46. DeleteOrder
47. UpdateReceived
48. UpdateValue
49. AddDishToOrder
50. UpdateDishQuantity
51. AddRefund
52. UpdateStatus
53. UpdateCollectiveInvoiceStatus
54. CancelReceipt
55. CancelOrder
56. CheckCollectiveInvoices

AddCustomer

Procedura dodaje klientów indywidualnych i firmy do ich wspólnej tabeli.

```
CREATE PROCEDURE AddCustomer
    @CustomerType nvarchar (10) = 'individual',
    @Phone nvarchar(9) = 'forbidden',
    @Mail nvarchar(30) = 'forbidden',
    @CustomerID int OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddCustomer
            INSERT INTO Customers(CustomerType, Phone, Mail)
            VALUES (@CustomerType, @Phone, @Mail);
            SET @CustomerID = @@IDENTITY
        COMMIT TRAN TAddCustomer
    END TRY
```

```
    BEGIN CATCH
    ROLLBACK TRAN TAddCustomer
    DECLARE @msg NVARCHAR(2048) =
    'Blad dodania klienta:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateCustomerData

Procedura aktualizuje dane klienta.

```
CREATE PROCEDURE UpdateCustomerData
    @CustomerID int,
    @Phone nvarchar(9),
    @Mail nvarchar(30)
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateCustomerData
            IF @Phone IS NOT NULL
            BEGIN
                UPDATE Customers SET Phone = @Phone WHERE CustomerID =
@CustomerID
            END
            IF @Mail IS NOT NULL
            BEGIN
                UPDATE Customers SET Mail = @Mail WHERE CustomerID =
@CustomerID
            END
        COMMIT TRAN TUpdateCustomerData
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateCustomerData
    DECLARE @msg NVARCHAR(2048) =
    'Blad aktualizacji danych klienta:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateDataForDiscounts

Procedura uruchamiana triggerem po aktualizacji settled i received na 1 w Orders.
Aktualizuje statystyki klienta dotyczące jego zamówień.

```
CREATE PROCEDURE UpdateDataForDiscounts
    @CustomerID int,
```

```
    @OrderValue decimal(20, 2)
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateDataForDiscounts
            DECLARE @IsIndividual bit
            EXEC @IsIndividual = CheckIfIndividual @CustomerID
            IF (@IsIndividual = 1)
                EXEC UpdateIndCustomerOrdersData
                    @CustomerID,
                    @OrderValue
            ELSE
                EXEC UpdateCompanyOrdersData
                    @CustomerID,
                    @OrderValue
        COMMIT TRAN TUpdateDataForDiscounts
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateDataForDiscounts
    DECLARE @msg NVARCHAR(2048) =
    'Blad aktualizacji danych zamówień dla klienta:' + CHAR(13)+CHAR(10)
+ ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddIndividualCustomer

Procedura dodaje klienta indywidualnego.

```
CREATE PROCEDURE AddIndividualCustomer
      @CompanyID int = null,
      @FirstName nvarchar(24) = 'forbidden',
      @LastName nvarchar(30) = 'forbidden',
      @Age int = NULL,
      @PersonalDateAgreement bit = 0,
      @NumberOfOrders int =0,
      @OrdersInRow int = 0,
      @SumOfOrders decimal(20,2) = 0,
      @CustomerType nvarchar(10) = 'individual',
      @Phone nvarchar(9) = 'forbidden',
      @Mail nvarchar(30) = 'forbidden',
      @CustomerID int OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
```

```
        BEGIN TRAN TAddIndCustomer
            EXEC AddCustomer
            @CustomerType,
            @Phone,
            @Mail,
            @CustomerID = @CustomerID OUTPUT
                  SET IDENTITY_INSERT dbo.IndividualCustomers ON
            INSERT INTO IndividualCustomers(IndividualCustomerID,
CompanyID, FirstName, LastName, Age, PersonalDataAgreement,
NumberOfOrders, OrdersInRow, SumOfOrders)
            VALUES (@CustomerID, @CompanyID, @FirstName, @LastName,
@Age, @PersonalDateAgreement, @NumberOfOrders, @OrdersInRow,
@SumOfOrders);
                  SET IDENTITY_INSERT dbo.IndividualCustomers OFF
            COMMIT TRAN TAddIndCustomer
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddIndCustomer
    DECLARE @msg NVARCHAR(2048) =
    'Blad dodania klienta indywidualnego:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateIndividualCustomerData

Procedura aktualizuje dane klienta indywidualnego.

```
CREATE PROCEDURE UpdateIndividualCustomerData
    @CustomerID int,
    @CompanyID int = NULL,
    @FirstName nvarchar(24),
    @LastName nvarchar(30),
    @Age int = NULL,
    @PersonalDataAgreement bit,
    @Phone nvarchar(9),
    @Mail nvarchar(30)
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateIndividualCustomerData
            IF @PersonalDataAgreement IS NOT NULL
            BEGIN
                UPDATE IndividualCustomers SET PersonalDataAgreement =
@PersonalDataAgreement WHERE IndividualCustomerID = @CustomerID
            END
            IF @PersonalDataAgreement = 1
            BEGIN
                IF @CompanyID IS NOT NULL
                BEGIN
                    UPDATE IndividualCustomers SET CompanyID = @CompanyID WHERE
```

```
IndividualCustomerID  = @CustomerID
                END
                IF @FirstName IS NOT NULL
                BEGIN
                    UPDATE IndividualCustomers SET FirstName = @FirstName WHERE
IndividualCustomerID  = @CustomerID
                END
                IF @LastName IS NOT NULL
                BEGIN
                    UPDATE IndividualCustomers SET LastName = @LastName WHERE
IndividualCustomerID  = @CustomerID
                END
                IF @Age IS NOT NULL
                BEGIN
                    UPDATE IndividualCustomers SET Age = @Age WHERE IndividualCustomerID
= @CustomerID
                END
                IF (@Phone IS NOT NULL OR @Mail IS NOT NULL)
                    BEGIN
                    EXEC UpdateCustomerData
                    @CustomerID,
                    @Phone,
                    @Mail
                    END
            END
        COMMIT TRAN TUpdateIndividualCustomerData
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateIndividualCustomerData
    DECLARE @msg NVARCHAR(2048) =
    'Blad aktualizacji danych klienta indywidualnego:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateIndCustomerOrdersData

Procedura uruchamiana po odebraniu i zapłacie za każde zamówienie klienta indywidualnego, aktualizuje jego statystyki dotyczące zamówień, a gdy spełnia on wtedy odpowiednie założenia, uruchamia procedurę przyznania odpowiedniego rabatu.

```
CREATE PROCEDURE UpdateIndCustomerOrdersData
@CustomerID int,
@OrderValue decimal(20, 2)
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateIndCustomerOrdersData

            UPDATE IndividualCustomers SET SumOfOrders += @OrderValue
WHERE IndividualCustomerID = @CustomerID
            IF(@OrderValue >= 30)
            BEGIN
                UPDATE IndividualCustomers SET NumberOfOrders += 1 WHERE
```

```
IndividualCustomerID = @CustomerID
                UPDATE IndividualCustomers SET OrdersInRow += 1 WHERE
IndividualCustomerID = @CustomerID
            END
            ELSE
            BEGIN
                UPDATE IndividualCustomers SET OrdersInRow = 0 WHERE
IndividualCustomerID = @CustomerID
            END

            IF((SELECT NumberOfOrders FROM IndividualCustomers WHERE
IndividualCustomerID = @CustomerID) = 10)
                EXEC AddGrantedDiscount
                    @CustomerID,
                    @DiscountID  = 1,
                    @DiscountValue = 0.03,
                    @ExpirationDate = NULL,
                    @Used = 0

            IF((SELECT OrdersInRow FROM IndividualCustomers WHERE
IndividualCustomerID = @CustomerID) = 10)
                EXEC AddGrantedDiscount
                    @CustomerID,
                    @DiscountID  = 2,
                    @DiscountValue = 0.03,
                    @ExpirationDate = NULL,
                    @Used = 0

            DECLARE @SumOfOrders int = (SELECT SumOfOrders FROM
IndividualCustomers WHERE IndividualCustomerID = @CustomerID)

                  DECLARE @NewDate datetime = DATEADD(DAY, 7, GETDATE())

            IF( @SumOfOrders >= 1000)
            BEGIN
                EXEC AddGrantedDiscount
                    @CustomerID,
                      @DiscountID  = 3,
                      @DiscountValue = 0.05,
                    @ExpirationDate = @NewDate,
                      @Used = 0
            END

            IF( @SumOfOrders >= 5000)
            BEGIN
                EXEC AddGrantedDiscount
                 @CustomerID,
                 @DiscountID  = 4,
```

```
                @DiscountValue = 0.05,
                @ExpirationDate = @NewDate,
                @Used = 0
            END

        COMMIT TRAN TUpdateIndCustomerOrdersData
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateIndCustomerOrdersData
    DECLARE @msg NVARCHAR(2048) =
    'Blad aktualizacji danych zamówień dla klienta indywidualnego:' +
CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddCompany

Procedura dodaje firmę.

```
CREATE PROCEDURE AddCompany
      @CompanyName nvarchar(30),
      @NIP nvarchar(10),
      @Country nvarchar(40),
      @ZipCode nvarchar(30),
      @Address nvarchar(40),
      @City nvarchar(40),
      @InvoicePeriod int,
      @NumberOfOrdersInMonth int = 0,
      @SumOfOrdersInMonth int = 0,
      @NumberOfMonthsInRow int = 0,
      @SumOfOrdersInQuarter decimal(20,2) = 0,
      @Phone nvarchar(9) = 'forbidden',
      @Mail nvarchar(30) = 'forbidden',
      @CustomerID int OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddCompany
            EXEC AddCustomer
            'company',
            @Phone,
            @Mail,
            @CustomerID = @CustomerID OUTPUT
                  SET IDENTITY_INSERT Companies ON
            INSERT INTO Companies(CompanyID, CompanyName, NIP, Country,
ZipCode, Address, City, InvoicePeriod, NumberOfOrdersInMonth,
```

```
NumberOfMonthsInRow, SumOfOrdersInQuarter)
            VALUES (@CustomerID, @CompanyName, @NIP, @Country, @ZipCode,
@Address, @City, @InvoicePeriod, @NumberOfOrdersInMonth,
@NumberOfMonthsInRow, @SumOfOrdersInQuarter);
                  SET IDENTITY_INSERT Companies OFF
            COMMIT TRAN TAddCompany
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddCompany
    DECLARE @msg NVARCHAR(2048) =
    'Blad dodania firmy:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateCompanyData

Procedura aktualizuje dane firmy.

```
CREATE PROCEDURE UpdateCompanyData
    @CustomerID int,
    @CompanyName nvarchar(30),
    @NIP nvarchar(10),
    @Country nvarchar(40),
    @ZipCode nvarchar(30),
    @Address nvarchar(40),
    @City nvarchar(40),
    @InvoicePeriod int,
    @Phone nvarchar(9),
    @Mail nvarchar(30)
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateCompanyData
            IF @CompanyName IS NOT NULL
            BEGIN
               UPDATE Companies SET CompanyName = @CompanyName WHERE CompanyID
= @CustomerID
            END
            IF @NIP IS NOT NULL
            BEGIN
                UPDATE Companies SET NIP = @NIP WHERE CompanyID  = @CustomerID
            END
            IF @Country IS NOT NULL
            BEGIN
                UPDATE Companies SET Country = @Country WHERE CompanyID  =
@CustomerID
            END
            IF @ZipCode IS NOT NULL
            BEGIN
                UPDATE Companies SET ZipCode = @ZipCode WHERE CompanyID  =
```

```
@CustomerID
            END
            IF @Address IS NOT NULL
            BEGIN
                UPDATE Companies SET Address = @Address WHERE CompanyID  =
@CustomerID
            END
            IF @City IS NOT NULL
            BEGIN
                UPDATE Companies SET City = @City WHERE CompanyID  = @CustomerID
            END
            IF @InvoicePeriod IS NOT NULL
            BEGIN
              UPDATE Companies SET InvoicePeriod = @InvoicePeriod WHERE
CompanyID  = @CustomerID
            END
            IF (@Phone IS NOT NULL OR @Mail IS NOT NULL)
                BEGIN
                EXEC UpdateCustomerData
                @CustomerID,
                @Phone,
                @Mail
                END
        COMMIT TRAN TUpdateCompanyData
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateCompanyData
    DECLARE @msg NVARCHAR(2048) =
    'Blad aktualizacji danych firmy:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateCompanyOrdersData

Procedura uruchamiana po odebraniu i zapłacie za każde zamówienie firmy, aktualizuje jej statystyki dotyczące zamówień.

```
CREATE PROCEDURE UpdateCompanyOrdersData
@CustomerID int,
@OrderValue decimal(20,2)
AS
BEGIN
  BEGIN TRY
   BEGIN TRAN UpdateCompanyOrdersData
    UPDATE Companies SET NumberOfOrdersInMonth += 1 WHERE CompanyID = @CustomerID
    UPDATE Companies SET SumOfOrdersInMonth += @OrderValue WHERE CompanyID = @CustomerID
    UPDATE Companies SET SumOfOrdersInQuarter += @OrderValue WHERE CompanyID =
@CustomerID

    COMMIT TRAN UpdateCompanyOrdersData
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN UpdateCompanyOrdersData
```

```
    DECLARE @msg NVARCHAR(2048) =
    'Błąd aktualizacji danych zamówień dla firmy:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

MonthlyCompanyUpdate

Procedura uruchamiana na początku każdego miesiąca, sprawdza statystyki zamówień firmy i przyznaje rabaty w przypadku spełenia wymagań. Na koniec resetuje statystyki.

```
CREATE PROCEDURE MonthlyCompanyUpdate
AS
BEGIN
    WAITFOR TIME '00:00';
    BEGIN TRY
    IF DAY(GETDATE()) = 1
    BEGIN
        BEGIN TRAN TMonthlyCompanyUpdate
        --iteruje po wszystkich firmach
            DECLARE @CompanyID int = 0
            WHILE (1=1)
            BEGIN
                SELECT TOP 1 @CompanyID = CompanyID
                FROM Companies
                WHERE CompanyID > @CompanyID
                ORDER BY CompanyID
                IF @@ROWCOUNT = 0 BREAK;

                --naliczanie rabatu miesięcznego
                IF ((SELECT NumberOfOrdersInMonth FROM Companies WHERE CompanyID = @CompanyID) >= 5
AND (SELECT SumOfOrdersInMonth FROM Companies WHERE CompanyID = @CompanyID) >= 500)
                BEGIN
                    UPDATE Companies SET NumberOfMonthsInRow += 1 WHERE CompanyID = @CompanyID
                        DECLARE @MonthsInRow int
                        SELECT @MonthsInRow = NumberOfMonthsInRow FROM Companies WHERE CompanyID =
@CompanyID
                        DECLARE @NewDiscountValue decimal(6,2) = 0.01 * @MonthsInRow
                        IF(@NewDiscountValue > 0.05) SET @NewDiscountValue = 0.05
                    BEGIN
                        DECLARE @NewDate datetime = DATEADD(month, 1, GETDATE())
                        EXEC AddGrantedDiscount
                          @CustomerID = @CompanyID,
                          @DiscountID  = 5,
                          @DiscountValue =     @NewDiscountValue,
                          @ExpirationDate = @NewDate,
                          @Used = 0
                    END
                END
                ELSE
                    UPDATE Companies SET NumberOfMonthsInRow = 0 WHERE CompanyID = @CompanyID
                --zeruje wartosc ze starego miesiąca
                UPDATE Companies SET NumberOfOrdersInMonth = 0 WHERE CompanyID = @CompanyID
                UPDATE Companies SET SumOfOrdersInMonth = 0 WHERE CompanyID = @CompanyID

                --drugi rabat (kwartalny)
                IF(MONTH(GETDATE()) IN (1, 4, 7, 10))
                BEGIN
                  DECLARE @SumOfOrdersInQuarter decimal(6,2)
                  SELECT @SumOfOrdersInQuarter = SumOfOrdersInQuarter FROM Companies WHERE CompanyID
= @CompanyID
                    SET @NewDiscountValue = 0.05 * @SumOfOrdersInQuarter
                    IF(@SumOfOrdersInQuarter >= 10000)
                    BEGIN
```

```
                        SET @NewDate = DATEADD(month, 3, GETDATE())
                        EXEC AddGrantedDiscount
                            @CustomerID = @CompanyID,
                            @DiscountID  = 6,
                            @DiscountValue = @NewDiscountValue,
                            @ExpirationDate = @NewDate,
                            @Used = 0
                    END
                UPDATE Companies SET SumOfOrdersInQuarter = 0 WHERE CompanyID = @CompanyID
               END

            END
        COMMIT TRAN TMonthlyCompanyUpdate
    END
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TMonthlyCompanyUpdate
    DECLARE @msg NVARCHAR(2048) =
    'Bład comiesiecznej aktualizacji:' + CHAR(13) + CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddEmployee

Procedura dodaje pracownika.

```
CREATE PROCEDURE AddEmployee
    @FirstName nvarchar(30),
    @LastName nvarchar(30),
    @Pesel nvarchar(11),
    @BirthDate date,
    @HireDate date,
    @Address nvarchar(60),
    @Phone nvarchar(9),
    @Mail nvarchar(30),
    @Post nvarchar(20),
    @Salary decimal(20,2) = 2600
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddEmployee
            INSERT INTO Employees(FirstName, LastName, Pesel, BirthDate,
HireDate, Address, Phone, Mail, Post, Salary)
            VALUES (@FirstName, @LastName, @Pesel, @BirthDate,
@HireDate, @Address, @Phone, @Mail, @Post, @Salary);
        COMMIT TRAN TAddEmployee
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddEmployee
    DECLARE @msg NVARCHAR(2048) =
    'Bład dodania pracownika:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
```

```
    END CATCH
END
GO
```

UpdateEmployeeData

Procedura aktualizuje dane pracownika.

```
CREATE PROCEDURE AddEmployee
    @FirstName nvarchar(30),
    @LastName nvarchar(30),
    @Pesel nvarchar(11),
    @BirthDate date,
    @HireDate date,
    @Address nvarchar(60),
    @Phone nvarchar(9),
    @Mail nvarchar(30),
    @Post nvarchar(20),
    @Salary decimal(20,2) = 2600
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddEmployee
            INSERT INTO Employees(FirstName, LastName, Pesel, BirthDate,
HireDate, Address, Phone, Mail, Post, Salary)
            VALUES (@FirstName, @LastName, @Pesel, @BirthDate,
@HireDate, @Address, @Phone, @Mail, @Post, @Salary);
        COMMIT TRAN TAddEmployee
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddEmployee
    DECLARE @msg NVARCHAR(2048) =
    'Bład dodania pracownika:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddDiscount

Procedura dodaje zniżkę.

```
CREATE PROCEDURE AddDiscount
    @CustomerType nvarchar (10),
    @DiscountValue decimal(6, 2),
    @DiscountType nvarchar(11)
AS
BEGIN
    SET NOCOUNT ON;
```

```
    BEGIN TRY
        BEGIN TRAN TAddDiscount
            INSERT INTO Discounts(CustomerType, DiscountValue,
DiscountType)
            VALUES (@CustomerType, @DiscountValue, @DiscountType);
        COMMIT TRAN TAddDiscount
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddDiscount
    DECLARE @msg NVARCHAR(2048) =
    'Błąd dodania znizki:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddGrantedDiscount

Procedura dodaje przyznaną klientowi zniżkę

```
CREATE PROCEDURE AddGrantedDiscount
      @CustomerID int,
      @DiscountID int,
      @DiscountValue decimal(6, 2),
      @ExpirationDate datetime = NULL,
      @Used bit = 0
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddGrantedDiscount
            DECLARE @GrantedDate datetime = GETDATE()
            INSERT INTO GrantedDiscounts(CustomerID, DiscountID,
DiscountValue, GrantedDate, ExpirationDate, Used)
            VALUES (@CustomerID, @DiscountID, @DiscountValue,
@GrantedDate, @ExpirationDate, @Used);
        COMMIT TRAN TAddGrantedDiscount
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddGrantedDiscount
    DECLARE @msg NVARCHAR(2048) =
    'Błąd dodania przyznanej znizki:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateUsed

Procedura aktualizuje zużycie zniżki jednorazowej.

```
CREATE PROCEDURE UpdateUsed
    @GrantedDiscountID int
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateUsed
            UPDATE GrantedDiscounts SET Used = 1 WHERE GrantedDiscountID
= @GrantedDiscountID
        COMMIT TRAN TUpdateUsed
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateUsed
    DECLARE @msg NVARCHAR(2048) =
    'Bład aktualizacji wykorzystania rabatu:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateExpiredValues

Procedura codziennie sprawdza przeterminowanie zniżek czasowych i zmienia ich wartość Used na 1.

```
CREATE PROCEDURE UpdateExpiredDiscounts
AS
BEGIN
    WAITFOR TIME '00:00';
    BEGIN TRY
    BEGIN TRAN TUpdateExpiredDiscounts
        DECLARE @GrantedDiscountID int = 0
        --pętla przechodzi po wszystkich wierszach i sprawdza, czy jakiś
rabat się przedawnił
        WHILE (1=1)
        BEGIN
            SELECT TOP 1 @GrantedDiscountID = GrantedDiscountID
            FROM GrantedDiscounts
            WHERE GrantedDiscountID > @GrantedDiscountID
            ORDER BY GrantedDiscountID
            IF @@ROWCOUNT = 0 BREAK;

                  DECLARE @ExpDate datetime
                  SELECT @ExpDate = ExpirationDate FROM GrantedDiscounts
WHERE GrantedDiscountID = @GrantedDiscountID
            IF ((@ExpDate IS NOT NULL) AND (@ExpDate < GETDATE()))
```

```
                EXEC UpdateUsed @GrantedDiscountID

        END
    COMMIT TRAN TUpdateExpiredDiscounts
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateExpiredDiscounts
    DECLARE @msg NVARCHAR(2048) =
    'Bład aktualizacji przedawnionych rabatow:' + CHAR(13) + CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

GetDiscountValue

Procedura dla przekazanego jej klienta przekazuje na wyjściu sumę wszystkich jego aktywnych zniżek procentowych.

```
CREATE PROCEDURE GetDiscountValue
    @CustomerID int,
    @result decimal(6,2) = 0 OUTPUT
AS
BEGIN
    BEGIN TRY
    BEGIN TRAN TGetDiscountValue
        DECLARE @GrantedDiscountID int = 0
        --pętla przechodzi po wszystkich wierszach i sprawdza, czy jakiś
rabat się przedawnił
        WHILE (1=1)
        BEGIN
            SELECT TOP 1 @GrantedDiscountID = GrantedDiscountID
            FROM GrantedDiscounts
            WHERE GrantedDiscountID > @GrantedDiscountID
            ORDER BY GrantedDiscountID
            IF @@ROWCOUNT = 0 BREAK;

            IF ((SELECT CustomerID FROM GrantedDiscounts WHERE
GrantedDiscountID = @GrantedDiscountID)  = @CustomerID)
            BEGIN
                DECLARE @DiscountID int = (SELECT DiscountID FROM
GrantedDiscounts WHERE GrantedDiscountID = @GrantedDiscountID)
                IF ( @DiscountID IN (3, 4))
                    EXEC UpdateUsed @GrantedDiscountID
                IF ( @DiscountID IN (1, 2, 3, 4, 5))
                    SET @result = @result + (SELECT DiscountValue FROM
GrantedDiscounts WHERE GrantedDiscountID = @GrantedDiscountID)
            END
```

```
        END
    COMMIT TRAN TGetDiscountValue
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TGetDiscountValue
    DECLARE @msg NVARCHAR(2048) =
    'Bład zwracania sumy rabatow:' + CHAR(13) + CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddTable

Procedura dodaje stolik.

```
CREATE PROCEDURE AddTable
    @NumberOfSeats int
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddTable
            INSERT INTO Tables(NumberOfSeats)
            VALUES (@NumberOfSeats);
        COMMIT TRAN TAddTable
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddTable
    DECLARE @msg NVARCHAR(2048) =
    'Bład dodania stolika:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

UpdateAvailability

Aktualizuje dostępność stolika.

```
CREATE PROCEDURE UpdateAvailability
    @TableID int,
    @Available bit -- 0 - niedostępny / 1 - dostępny
AS
BEGIN
    BEGIN TRY
        BEGIN TRAN TUpdateAvailability
            UPDATE Tables SET Availability = @Available WHERE TableID =
@TableID
```

```
        COMMIT TRAN TUpdateAvailability
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TUpdateAvailability
    DECLARE @msg NVARCHAR(2048) =
    'Bład aktualizacji dostepnosci stolika:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddReservation

Procedura uruchamia funkcję sprawdzającą, czy klient spełnia wymagania na złożenie rezerwacji. Jeśli tak,to rezerwacja zostaje dodana.

```
CREATE PROCEDURE AddReservation
    @OrderID int,
    @CustomerID int,
    @NumberOfPeople int
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddReservation
            DECLARE @CanMakeReservation bit

            EXEC @CanMakeReservation = CheckReservationConditions
@CustomerID, @OrderID, @NumberOfPeople
            IF @CanMakeReservation = 1
                INSERT INTO Reservations(OrderID, CustomerID,
NumberOfPeople)
                VALUES (@OrderID, @CustomerID, @NumberOfPeople);
        COMMIT TRAN TAddReservation
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddReservation
    DECLARE @msg NVARCHAR(2048) =
    'Bład dodania rezerwacji:' + CHAR(13)+CHAR(10) + ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddTableToReservation

Procedura dodająca stoliki do rezerwacji.

```
CREATE PROCEDURE AddTableToReservation
    @ReservationID int,
    @TableID int
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN TAddTableToReservation
            INSERT INTO ReservationDetails(ReservationID, TableID)
            VALUES (@ReservationID, @TableID);
        COMMIT TRAN TAddTableToReservation
    END TRY
    BEGIN CATCH
    ROLLBACK TRAN TAddTableToReservation
    DECLARE @msg NVARCHAR(2048) =
    'Bład dodania stolika do rezerwacji:' + CHAR(13)+CHAR(10) +
ERROR_MESSAGE();
    THROW 52000,@msg, 1;
    END CATCH
END
GO
```

AddCategory

Dodaje kategorię produktów do bazy.

```
CREATE PROCEDURE [dbo].[AddCategory]
      @CategoryName nvarchar(30)
AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TAddCategory
                  INSERT INTO Categories(CategoryName)
                  VALUES (@CategoryName);
            COMMIT TRAN TAddCategory
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TAddCategory
            DECLARE @msg nvarchar(2048) =
            'Blad dodania kategorii:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddMenu

Dodaje nowe menu do bazy.

```
CREATE PROCEDURE [dbo].[AddMenu]
	@InseritonDate datetime = NULL,
	@RemovalDate datetime = NULL

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TAddMenu
			IF (@InseritonDate IS NULL)
			BEGIN
				SET @InseritonDate = (SELECT DATEADD(DAY, 1,
GETDATE()));
			END
			IF (@RemovalDate IS NULL OR @RemovalDate > DATEADD(WEEK,
2, @InseritonDate))
			BEGIN
				SET @RemovalDate = DATEADD(WEEK, 2,
@InseritonDate);
			END
			IF (@InseritonDate < (SELECT DATEADD(DAY, 1, GETDATE())))
			BEGIN
				;THROW 52000, 'Menu nie moze byc wprowadzone z
wyprzedzeniem mniejszym niz 1 dzien', 1;
			END
			ELSE
			BEGIN
				DECLARE @CurrentMenu int
				EXEC @CurrentMenu = dbo.GetCurrentMenu
				IF (@CurrentMenu IS NOT NULL)
				BEGIN
					EXEC dbo.RemoveMenu @CurrentMenu,
@InseritonDate
				END
				INSERT INTO Menu(InsertionDate, RemovalDate)
				VALUES (@InseritonDate, @RemovalDate);
			END
		COMMIT TRAN TAddMenu
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TAddMenu
		DECLARE @msg nvarchar(2048) =
		'Blad utworzenia menu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

RemoveMenu

Usuwa menu z bazy.

```
CREATE PROCEDURE [dbo].[RemoveMenu]
	@MenuID int,
	@RemovalDate datetime = NULL

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TRemoveMenu
			IF (@RemovalDate IS NULL)
			BEGIN
				SET @RemovalDate = GETDATE();
			END
			IF (@RemovalDate <= (SELECT RemovalDate FROM Menu WHERE
MenuID = @MenuID))
			BEGIN
				UPDATE Menu
				SET RemovalDate = @RemovalDate
				WHERE MenuID = @MenuID;
			END
			ELSE
			BEGIN
				;THROW 52000, 'Menu nie moze byc usuniete pozniej
niz bylo to poczatkowo zakladane', 1;
			END
		COMMIT TRAN TRemoveMenu
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TRemoveMenu
		DECLARE @msg nvarchar(2048) =
		'Blad usuniecia menu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

AddDishToMenu

Dodaje danie do menu.

```
CREATE PROCEDURE [dbo].[AddDishToMenu]
	@MenuID int,
	@DishID int,
	@DishPrice decimal(20,2) = NULL,
	@Availability bit = 1

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
```

```
            BEGIN TRAN TAddDishToMenu
                  DECLARE @CanAdd bit
                  EXEC @CanAdd = dbo.CanAddDishToMenu @DishID, @MenuID
                  IF (@DishPrice IS NULL)
                  BEGIN
                        SET @DishPrice = (SELECT DishPrice FROM Dishes
WHERE DishID = @DishID)
                  END
                  IF ((SELECT COUNT(*) FROM MenuDetails WHERE MenuID =
@MenuID AND DishID = @DishID) != 0)
                  BEGIN
                        ;THROW 52000, 'Danie juz widnieje w menu', 1;
                  END
                  ELSE IF(@CanAdd = 0)
                  BEGIN
                        ;THROW 52000, 'Danie nie spelnia warunkow', 1;
                  END
                  ELSE
                  BEGIN
                        INSERT INTO MenuDetails(MenuID, DishID, DishPrice,
Availability)
                        VALUES (@MenuID, @DishID, @DishPrice,
@Availability)
                  END
            COMMIT TRAN TAddDishToMenu
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TAddDishToMenu
            DECLARE @msg nvarchar(2048) =
            'Blad dodawania dania do
menu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

UpdateDishAvailability

Zmienia dostępność dania.

```
CREATE PROCEDURE [dbo].[UpdateDishAvailability]
      @MenuID int,
      @DishID int,
      @NewAvailability bit

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TUpdateDishAvailability
```

```
                  UPDATE MenuDetails
                  SET Availability = @NewAvailability
                  WHERE MenuID = @MenuID AND DishID = @DishID
            COMMIT TRAN TUpdateDishAvailability
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TUpdateDishAvailability
            DECLARE @msg nvarchar(2048) =
            'Blad zmiany dostenosci dania w
menu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddProduct

Dodaje nowy produkt do bazy.

```
CREATE PROCEDURE [dbo].[AddProduct]
      @ProductName nvarchar(50),
      @CategoryID int,
      @UnitPrice decimal(20,2),
      @UnitsInStock smallint = 0

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TAddProduct
                  INSERT INTO Products(ProductName, CategoryID, UnitPrice,
UnitsInStock)
                  VALUES (@ProductName, @CategoryID, @UnitPrice,
@UnitsInStock);
            COMMIT TRAN TAddProduct
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TAddProduct
            DECLARE @msg nvarchar(2048) =
            'Blad dodania Produktu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

UpdateUnitsInStock

Zmienia aktualny stan magazynowy produktu.

```
CREATE PROCEDURE [dbo].[UpdateUnitsInStock]
```

```
	@ProductID int,
	@ChangedUnits int

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TUpdateUnitsInStock
			DECLARE @CurrentUnits int, @CurrentMenu int,
@UpcomingMenu int
			UPDATE Products
			SET UnitsInStock = UnitsInStock + @ChangedUnits
			WHERE ProductID = @ProductID;

			SET @CurrentUnits = (SELECT UnitsInStock FROM Products
WHERE ProductID = @ProductID)
			SET @CurrentMenu = dbo.GetCurrentMenu()
			SET @UpcomingMenu = dbo.GetUpcomingMenu()

			IF @ChangedUnits > 0
			--dostawa produktu
			BEGIN
				DECLARE @MyCursor CURSOR, @MenuID int, @DishID int
				SET @MyCursor = CURSOR FOR
				SELECT MD.MenuID, MD.DishID FROM MenuDetails MD
				INNER JOIN DishDetails DD
				ON MD.DishID = DD.DishID
				WHERE(MD.MenuID = @CurrentMenu OR MD.MenuID =
@UpcomingMenu)
				AND MD.Availability = 0 AND DD.ProductID =
@ProductID
				AND DD.Quantity <= @CurrentUnits

				OPEN @MyCursor
				FETCH NEXT FROM @MyCursor
				INTO @MenuID, @DishID

				WHILE @@FETCH_STATUS = 0
				BEGIN
					EXEC dbo.UpdateDishAvailability @MenuID,
@DishID, 1
					FETCH NEXT FROM @MyCursor
					INTO @MenuID, @DishID
				END;

				CLOSE @MyCursor ;
				DEALLOCATE @MyCursor;
			END;
			ELSE IF @ChangedUnits < 0
			BEGIN
```

```
                        SET @MyCursor = CURSOR FOR
                        SELECT MD.MenuID, MD.DishID FROM MenuDetails MD
                        INNER JOIN DishDetails DD
                        ON MD.DishID = DD.DishID
                        WHERE(MD.MenuID = @CurrentMenu OR MD.MenuID =
@UpcomingMenu)
                        AND MD.Availability = 1 AND DD.ProductID =
@ProductID
                        AND DD.Quantity > @CurrentUnits

                        OPEN @MyCursor
                        FETCH NEXT FROM @MyCursor
                        INTO @MenuID, @DishID

                        WHILE @@FETCH_STATUS = 0
                        BEGIN
                              EXEC dbo.UpdateDishAvailability @MenuID,
@DishID, 0
                              FETCH NEXT FROM @MyCursor
                              INTO @MenuID, @DishID
                        END;

                        CLOSE @MyCursor ;
                        DEALLOCATE @MyCursor;
                  END;
            COMMIT TRAN TUpdateUnitsInStock
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TUpdateUnitsInStock
            DECLARE @msg nvarchar(2048) =
            'Blad zmiany dostepnej ilosci
produktu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

UpdateUnitPrice

Zmienia aktualną cenę jednostkową produktu.

```
CREATE PROCEDURE [dbo].[UpdateUnitPrice]
      @ProductID int,
      @NewPrice decimal(20,2)

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TUpdateUnitPrice
```

```
                  UPDATE Products
                  SET UnitPrice = @NewPrice
                  WHERE ProductID = @ProductID;
            COMMIT TRAN TUpdateUnitPrice
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TUpdateUnitPrice
            DECLARE @msg nvarchar(2048) =
            'Blad zmiany ceny produktu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddDish

Dodaje nowe danie do bazy.

```
CREATE PROCEDURE [dbo].[AddDish]
      @DishName nvarchar(40),
      @DishPrice decimal(20,2) = 0,
      @RemovalDate datetime = NULL

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TAddDish
                  INSERT INTO Dishes (DishName, DishPrice, RemovalDate)
                  VALUES (@DishName, @DishPrice, @RemovalDate);
            COMMIT TRAN TAddDish
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TAddDish
            DECLARE @msg nvarchar(2048) =
            'Blad utworzenia dania:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

UpdateDishPrice

Zmienia aktualną cenę dania.

```
CREATE PROCEDURE [dbo].[UpdateDishPrice]
      @DishID int,
      @NewPrice int

AS
```

```
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TUpdateDishPrice
			UPDATE Dishes
			SET DishPrice = @NewPrice
			WHERE DishID = @DishID;
		COMMIT TRAN TUpdateDishPrice
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TUpdateDishPrice
		DECLARE @msg nvarchar(2048) =
		'Blad zmiany ceny dania:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

RemoveDish

Ustawia datę usunięcia dania z menu.

```
CREATE PROCEDURE [dbo].[RemoveDish]
	@DishID int

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TRemoveDish
			UPDATE Dishes
			SET RemovalDate = GETDATE()
			WHERE DishID = @DishID;
		COMMIT TRAN TRemoveDish
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TRemoveDish
		DECLARE @msg nvarchar(2048) =
		'Blad wycofania dania:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

AddProductToDish

Dodaje produkt do dania.

```
CREATE PROCEDURE [dbo].[AddProductToDish]
	@DishID int,
```

```
	@ProductID int,
	@UnitPrice decimal(20,2),
	@Quantity int = 1

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TProductToDish
			INSERT INTO DishDetails (DishID, ProductID, UnitPrice,
Quantity)
			VALUES (@DishID, @ProductID, @UnitPrice, @Quantity);

			DECLARE @DishValue decimal(20,2) = @UnitPrice*@Quantity
			DECLARE @CurrentValue decimal(20,2)
			EXEC @CurrentValue = GetCurrentDishPrice @DishID
			DECLARE @NewValue decimal(20,2) = @DishValue +
@CurrentValue

			EXEC dbo.UpdateDishPrice
				@DishID,
				@NewValue

		COMMIT TRAN TAddProduct
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TProductToDish
		DECLARE @msg nvarchar(2048) =
		'Blad dodania Produktu do
dania:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

AddReceipt

Dodaje rachunek za zamówienie.

```
CREATE PROCEDURE [dbo].[AddReceipt]
	@InvoicedCustomer int,
	@InvoiceDate datetime,
	@AccountNumber nvarchar(26),
	@ReceiptType nvarchar(20),
	@PaymentMethod nvarchar(8),
	@Value decimal(20,2),
	@Settled bit = 0,
	@Cancelled bit = 0,
	@SaleDate datetime,
	@ReceiptID int OUTPUT

AS
```

```
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TAddReceipt
			INSERT INTO Receipt(InvoicedCustomer, InvoiceDate, AccountNumber,
ReceiptType, PaymentMethod, Value, Settled, Cancelled, SaleDate)
			VALUES (@InvoicedCustomer, @InvoiceDate, @AccountNumber,
@ReceiptType, @PaymentMethod, @Value, @Settled, @Cancelled, @SaleDate)
			SET @ReceiptID = @@IDENTITY
			IF (@ReceiptType = 'collective invoice')
				BEGIN
					DECLARE @InvoicePeriod int, @ValidTo datetime
					SET @InvoicePeriod = (SELECT InvoicePeriod FROM
Companies WHERE CompanyID = @InvoicedCustomer)
					SET @ValidTo = DATEADD(month, @InvoicePeriod,
@SaleDate)
					EXEC dbo.AddCollectiveInvoice @ReceiptID, @SaleDate,
@ValidTo, 1
				END
		COMMIT TRAN TAddReceipt
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TAddReceipt
		DECLARE @msg nvarchar(2048) =
		'Blad dodawania rachunku:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

CancelReceipt

Anuluje rachunek.

```
CREATE PROCEDURE [dbo].[CancelReceipt]
	@ReceiptID int,
	@OrderID int

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TCancelReceipt
			IF (dbo.IsCollectiveInvoice (@ReceiptID) = 1)
			BEGIN
				EXEC dbo.RemoveOrderFromReceipt @ReceiptID,
@OrderID
			END
			ELSE IF (SELECT Settled FROM Receipt WHERE ReceiptID =
@ReceiptID) = 1
			BEGIN
				DECLARE @Value decimal(20,2)
				EXEC @Value = dbo.GetReceiptValue @ReceiptID
				EXEC dbo.AddRefund @ReceiptID, @OrderID, @Value
```

```
                  END
                  UPDATE Receipt
                  SET Cancelled = 1
                  WHERE ReceiptID = @ReceiptID
            COMMIT TRAN TCancelReceipt
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TCancelReceipt
            DECLARE @msg nvarchar(2048) =
            'Blad anulowania rachunku:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

ChangeReceiptValue

Zmienia wartość rachunku.

```
CREATE PROCEDURE [dbo].[ChangeReceiptValue]
      @ReceiptID int,
      @NewValue decimal(20,2)

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TChangeReceiptValue
                  UPDATE Receipt
                  SET Value = @NewValue
                  WHERE ReceiptID = @ReceiptID
            COMMIT TRAN TChangeReceiptValue
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TChangeReceiptValue
            DECLARE @msg nvarchar(2048) =
            'Blad zmiany wartosci
rachunku:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddOrderToReceipt

Dodaje zamówienie do rachunku.

```
CREATE PROCEDURE [dbo].[AddOrderToReceipt]
      @ReceiptID int,
      @OrderID int
```

```
AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TAddOrderToReceipt
			DECLARE @OrderValue decimal(20,2), @ReceiptValue
decimal(20,2), @NewValue decimal(20,2)
			EXEC @OrderValue = dbo.GetValueAfterDiscount @OrderID
			EXEC @ReceiptValue = dbo.GetReceiptValue @ReceiptID
			SET @NewValue = @ReceiptValue + @OrderValue
			EXEC dbo.ChangeReceiptValue @ReceiptID, @NewValue
			INSERT INTO ReceiptDetails(ReceiptID, OrderID)
			VALUES (@ReceiptID, @OrderID)
		COMMIT TRAN TAddOrderToReceipt
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TAddOrderToReceipt
		DECLARE @msg nvarchar(2048) =
		'Blad dodawania zamowienia do
rachunku:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

RemoveOrderFromReceipt

Usuwa zamówienie z rachunku.

```
CREATE PROCEDURE [dbo].[RemoveOrderFromReceipt]
	@ReceiptID int,
	@OrderID int

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TRemoveOrderFromReceipt
			DECLARE @OrderValue decimal(20,2), @ReceiptValue
decimal(20,2), @NewValue decimal(20,2)
			EXEC @OrderValue = dbo.GetValueAfterDiscount @OrderID
			EXEC @ReceiptValue = dbo.GetReceiptValue @ReceiptID
			SET @NewValue = @ReceiptValue - @OrderValue
			EXEC dbo.ChangeReceiptValue @ReceiptID, @NewValue
			DELETE FROM ReceiptDetails
			WHERE ReceiptID = @ReceiptID AND OrderID = @OrderID
		COMMIT TRAN TRemoveOrderFromReceipt
	END TRY
	BEGIN CATCH
```

```
            ROLLBACK TRAN TRemoveOrderFromReceipt
            DECLARE @msg nvarchar(2048) =
            'Blad dodawania zamowienia do
rachunku:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

SettleReceipt

Opłaca rachunek

```
CREATE PROCEDURE [dbo].[SettleReceipt]
      @ReceiptID int

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TSettleReceipt
                  DECLARE @MyCursor CURSOR
                  DECLARE @OrderID int
                  SET @MyCursor = CURSOR FOR
                  SELECT * FROM dbo.GetOrdersFromReceipt (@ReceiptID)

                  OPEN @MyCursor
                  FETCH NEXT FROM @MyCursor
                  INTO @OrderID

                  WHILE @@FETCH_STATUS = 0
                  BEGIN
                        EXEC dbo.SettleOrder @OrderID
                        FETCH NEXT FROM @MyCursor
                        INTO @OrderID
                  END;

                  CLOSE @MyCursor ;
                  DEALLOCATE @MyCursor;

                  UPDATE Receipt
                  SET Settled = 1
                  WHERE ReceiptID = @ReceiptID
            COMMIT TRAN TSettleReceipt
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TSettleReceipt
            DECLARE @msg nvarchar(2048) =
            'Blad oplacania rachunku:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
```

```
	END CATCH
END
GO
```

InvoiceReceipt

Wystawia rachunek do zamówienia, ktory następnie opłaci klient.

```
CREATE PROCEDURE [dbo].[InvoiceReceipt]
	@ReceiptID int,
	@InvoiceDate datetime

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TInvoiceReceipt
			UPDATE Receipt
			SET InvoiceDate = @InvoiceDate
			WHERE ReceiptID = @ReceiptID
		COMMIT TRAN TInvoiceReceipt
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TInvoiceReceipt
		DECLARE @msg nvarchar(2048) =
		'Blad wystawiania faktury:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

AddOrder

Dodaje zamówienie do bazy

```
CREATE PROCEDURE [dbo].[AddOrder]
	@CustomerID int,
	@EmployeeID int,
	@ReceptionType nvarchar(11) = 'takeaway',
	@PaymentType nvarchar(11),
	@OrderDate datetime = NULL, --data złożenia zamówienia
@ReceiveDate datetime , --tu ma być data z formularza, data odebrania
zamówienia
	@Value decimal(20,2) = 0,
	@Received bit = NULL,
	@Settled bit = 0,
	@OrderID int OUT

AS
BEGIN
```

```
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TAddOrder

			IF @OrderDate IS NULL
			BEGIN
				SET @OrderDate = GETDATE();
			END

			INSERT INTO Orders(CustomerID, EmployeeID, ReceptionType,
PaymentType, OrderDate, ReceiveDate, Value, Discount, Completed, Received,
Settled)
			VALUES (@CustomerID, @EmployeeID, @ReceptionType,
@PaymentType, @OrderDate, @ReceiveDate, @Value, NULL, 0, @Received,
@Settled)

			SET @OrderID = @@IDENTITY

		COMMIT TRAN TAddOrder
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TAddOrder
		DECLARE @msg nvarchar(2048) =
		'Blad dodawania zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

CancelOrder

Anuluje zamówienie

```
CREATE PROCEDURE [dbo].[CancelOrder]
	@OrderID int

AS
BEGIN
	SET NOCOUNT ON
	BEGIN TRY
		BEGIN TRAN TCancelOrder
			DECLARE @ReceiptID int
			EXEC @ReceiptID = dbo.GetOrderReceipt @OrderID
			IF (@ReceiptID IS NOT NULL)
			--zamowienie zostalo juz oplacone
			BEGIN
				EXEC dbo.CancelReceipt @ReceiptID, @OrderID
			END
			UPDATE Orders
			SET Received = 0
```

```
                  WHERE OrderID = @OrderID
            COMMIT TRAN TCancelOrder
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TCancelOrder
            DECLARE @msg nvarchar(2048) =
            'Blad anulowania
zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

DeleteOrder

Usuwa zamówienie z bazy - procedura wykorzystywana tylko w skrajnych przypadkach

```
CREATE PROCEDURE [dbo].[DeleteOrder]
      @OrderID int

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TDeleteOrder
                  DELETE FROM Orders
                  WHERE OrderID = @OrderID
            COMMIT TRAN TDeleteOrder
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TDeleteOrder
            DECLARE @msg nvarchar(2048) =
            'Blad usuwania zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

SettleOrder

Opłaca zamówienie

```
CREATE PROCEDURE [dbo].[SettleOrder]
      @OrderID int

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TSettleOrder
```

```
                  UPDATE Orders
                  SET Settled = 1
                  WHERE OrderID = @OrderID
            COMMIT TRAN TSettleOrder
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TSettleOrder
            DECLARE @msg nvarchar(2048) =
            'Blad rozliczania
zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddDiscountToOrder

Dodaje rabat do zamówienia

```
CREATE PROCEDURE [dbo].[AddDiscountToOrder]
      @OrderID int

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TAddDiscountToOrder
                  DECLARE @CustomerID int, @DiscountPercent decimal(6, 2) =
0.00,
                  @SixthDiscountValue decimal(6,2) = 0.00,
                  @SixthDiscountID int, @ReceiptID int,
                  @DiscountValue decimal(20,2) = 0.00, @OrderValue
decimal(20,2)
                  SET @CustomerID = dbo.GetCustomerFromOrder(@OrderID)
                  EXEC dbo.GetDiscountValue @CustomerID, @result =
@DiscountPercent OUTPUT;
                  (SELECT @SixthDiscountValue = DiscountValue,
@SixthDiscountID = GrantedDiscountID FROM
dbo.GetSixthDiscount(@CustomerID));
                  SET @OrderValue = dbo.GetOrderValue(@OrderID);

                  IF @DiscountPercent IS NULL
                  BEGIN
                        SET @DiscountPercent = 0.00
                  END

                  SET @DiscountValue = @DiscountPercent*@OrderValue

                  IF @SixthDiscountValue IS NOT NULL
                  BEGIN
```

```
                        IF ((@OrderValue - @DiscountValue -
@SixthDiscountValue) >= 0)
                        BEGIN
                              EXEC dbo.UpdateUsed @SixthDiscountID
                              SET @DiscountValue += @SixthDiscountValue
                        END
                  END

                  UPDATE Orders
                  SET Discount = @DiscountValue
                  WHERE OrderID = @OrderID

            COMMIT TRAN TUpdateReceived
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TUpdateReceived
            DECLARE @msg nvarchar(2048) =
            'Blad dodawania rabatu do
zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

InvoiceOrder

Wystawia rachunek na podstawie kryteriów zamówienia

```
CREATE PROCEDURE [dbo].[InvoiceOrder]
      @OrderID int,
      @AccountNumber nvarchar(26) = NULL

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TInvoiceOrder

                  DECLARE @CustomerID int, @ReceiptID int,
                              @DiscountValue decimal(20,2),
                              @PaymentType nvarchar(11),
                              @OrderDate datetime,
                              @SaleDate datetime,
                              @ParentCompany int,
                              @PaymentMethod nvarchar(8)
                  SET @CustomerID = (SELECT CustomerID
                                    FROM Orders WHERE
                                    OrderID = @OrderID)
                  SET @PaymentType = (SELECT PaymentType
                                     FROM Orders WHERE
                                     OrderID = @OrderID)
                  SET @SaleDate = GETDATE()
```

```
SET @ParentCompany = (SELECT CompanyID
                      FROM IndividualCustomers
                      WHERE IndividualCustomerID
                      = @CustomerID)

EXEC dbo.AddDiscountToOrder @OrderId

IF (dbo.CheckIfIndividual(@CustomerID) = 1
    AND @ParentCompany IS NULL)
--w pelni indywidualny klient
BEGIN
       IF (@PaymentType = 'in-advance')
       BEGIN
              SET @PaymentMethod = 'transfer'
       END
       ELSE
       --platnosc przy odbiorze
       BEGIN
              SET @PaymentMethod = 'cash'
       END
       EXEC dbo.AddReceipt
              @InvoicedCustomer = @CustomerID,
              @InvoiceDate = NULL,
              @AccountNumber = @AccountNumber,
              @ReceiptType = 'receipt',
              @PaymentMethod = @PaymentMethod,
              @Value = 0,
              @Settled = 0,
              @Cancelled = 0,
              @SaleDate = @SaleDate,
              @ReceiptID = @ReceiptID OUTPUT
       EXEC dbo.AddOrderToReceipt @ReceiptID, @OrderID
       EXEC dbo.InvoiceReceipt @ReceiptID, @SaleDate
END
ELSE IF (dbo.CheckIfIndividual (@CustomerID) = 1
         AND @ParentCompany IS NOT NULL)
--indywidualny klient firmowy
BEGIN
--na rachunku bedzie firma
       SET @CustomerID = @ParentCompany
END
IF (dbo.CheckIfIndividual (@CustomerID) = 0)
--klient firmowy
BEGIN
       DECLARE @InvoicePeriod int
       SET @InvoicePeriod = (SELECT InvoicePeriod
                             FROM Companies WHERE
                             CompanyID = @CustomerID)
       IF (@InvoicePeriod < 0)
       --jesli nie rozliczamy go na fakture zbiorcza
       BEGIN
              IF (@PaymentType = 'in-advance')
              BEGIN
                     SET @PaymentMethod = 'transfer'
              END
              ELSE
```

```
                                --platnosc przy odbiorze
                                BEGIN
                                        SET @PaymentMethod = 'cash'
                                END
                                EXEC dbo.AddReceipt
                                        @InvoicedCustomer = @CustomerID,
                                        @InvoiceDate = NULL,
                                        @AccountNumber = @AccountNumber,
                                        @ReceiptType = 'one-time invoice',
                                        @PaymentMethod = 'cash',
                                        @Value = 0,
                                        @Settled = 0,
                                        @Cancelled = 0,
                                        @SaleDate = @SaleDate,
                                        @ReceiptID = @ReceiptID OUTPUT
                                EXEC dbo.AddOrderToReceipt @ReceiptID, @OrderID
                                EXEC dbo.InvoiceReceipt @ReceiptID, @SaleDate
                        END
                        ELSE
                        --dodajemy do faktury zbiorczej
                        BEGIN
SET @ReceiptID = (SELECT TOP(1) R.InvoicedCustomer
                                FROM CollectiveInvoices CI
                                INNER JOIN Receipt R ON
                                CI.ReceiptID = R.ReceiptID
                                WHERE CI.InProgress = 1)
                                IF (@ReceiptID IS NULL)
                                --trzeba utworzyc nowa fakture
                                BEGIN
                                        EXEC dbo.AddReceipt
                                          @InvoicedCustomer = @CustomerID,
                                          @InvoiceDate = NULL,
                                          @AccountNumber = @AccountNumber,
                                          @ReceiptType = 'collective invoice',
                                          @PaymentMethod = 'transfer',
                                          @Value = 0,
                                          @Settled = 0,
                                          @Cancelled = 0,
                                          @SaleDate = @SaleDate,
                                          @ReceiptID = @ReceiptID OUTPUT

                                END
                                EXEC dbo.AddOrderToReceipt @ReceiptID, @OrderID
                        END
                END

                COMMIT TRAN TInvoiceOrder
        END TRY
        BEGIN CATCH
                ROLLBACK TRAN TInvoiceOrder
                DECLARE @msg nvarchar(2048) =
                'Blad rozliczania zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
                THROW 52000, @msg, 1;
        END CATCH
END
GO
```

CompleteOrder

Zatwierdza stan zamówienia - uruchamiana po dodaniu wszystkich dań do zamówienia

```
CREATE PROCEDURE [dbo].[CompleteOrder]
	@OrderID int

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TCompleteOrder
			DECLARE @PaymentType nvarchar(11)
			SET @PaymentType = (SELECT PaymentType FROM Orders WHERE
OrderID = @OrderID)
			IF @PaymentType = 'in-advance' OR
dbo.IsReceived(@OrderID) = 1
			BEGIN
				UPDATE Orders
				SET Completed = 1
				WHERE OrderID = @OrderID
				EXEC dbo.InvoiceOrder @OrderID
			END
			ELSE
			BEGIN
				;THROW 52000, 'Nie mozna zatwierdzic zamowienia
przy platnosci na miejscu, jesli zamowienie nie zostalo jeszcze odebrane',
1;
			END
		COMMIT TRAN TCompleteOrder
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TCompleteOrder
		DECLARE @msg nvarchar(2048) =
		'Blad zamykania zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

UpdateReceived

Zmienia stan zamówienia

```
CREATE PROCEDURE [dbo].[UpdateReceived]
	@OrderID int,
	@Received bit

AS
```

```
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TUpdateReceived
			UPDATE Orders
			SET Received = @Received
			WHERE OrderID = @OrderID
		COMMIT TRAN TUpdateReceived
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TUpdateReceived
		DECLARE @msg nvarchar(2048) =
		'Blad realizacji
zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

UpdateValue

Zmienia wartość zamówienia

```
CREATE PROCEDURE [dbo].[UpdateValue]
	@OrderID int,
	@NewValue decimal(20,2)

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TUpdateValue
			DECLARE @Completed bit
			SET @Completed = (SELECT Completed FROM Orders WHERE
OrderID = @OrderID)

			IF (@Completed = 0)
			--mozna zmienic wartosc zamowienia tylko gdy nie zostalo
zatwierdzone
			BEGIN
				UPDATE Orders
				SET Value = @NewValue
				WHERE OrderID = @OrderID
			END

			ELSE
			BEGIN
				;THROW 52000, 'Nie mozna zmienic zawartosci
zamowienia gdy zostalo juz zatwierdzone', 1;
			END
```

```
            COMMIT TRAN TUpdateValue
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TUpdateValue
            DECLARE @msg nvarchar(2048) =
            'Blad zmiany wartosci
zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddDishToOrder

Dodaje danie do zamówienia

```
CREATE PROCEDURE [dbo].[AddDishToOrder]
      @OrderID int,
      @DishID int,
      @Quantity smallint = 1

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TAddDishToOrder
                  DECLARE @UnitPrice decimal(20,2), @CurrentMenu int,
                              @CanBeAdded bit, @OrderValue decimal(20,2),
@NewValue decimal(20,2)
                  EXEC @CurrentMenu = dbo.GetCurrentMenu
                  EXEC @CanBeAdded = dbo.CanAddDishToOrder @DishID,
@OrderID
                  SET @UnitPrice = (SELECT DishPrice FROM MenuDetails WHERE
DishID = @DishID AND MenuID = @CurrentMenu)
                  SET @OrderValue = dbo.GetOrderValue(@OrderID)
                  IF (@CanBeAdded = 1)
                  BEGIN
                        INSERT INTO OrderDetails(OrderID, DishID,
UnitPrice, Quantity)
                        VALUES (@OrderID, @DishID, @UnitPrice, @Quantity);
                        SET @NewValue = @OrderValue + @UnitPrice*@Quantity
                        EXEC dbo.UpdateValue @OrderID, @NewValue
                  END
                  ELSE
                  BEGIN
                        ;THROW 52000, 'Danie nie spelnia warunkow', 1;
                  END
            COMMIT TRAN TAddDishToOrder
      END TRY
```

```
	BEGIN CATCH
		ROLLBACK TRAN TAddDishToOrder
		DECLARE @msg nvarchar(2048) =
		'Blad dodawania dania do
zamowienia:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

UpdateDishQuantity

Zmienia ilość danego dania w zamówieniu

```
CREATE PROCEDURE [dbo].[UpdateDishQuantity]
		@OrderID int,
	@DishID int,
	@NewQuantity smallint

AS
BEGIN
	SET NOCOUNT ON;
	BEGIN TRY
		BEGIN TRAN TUpdateDishQuantity
			DECLARE @OldQuantity smallint,
					@DishPrice decimal(20,2),
					@OrderValue decimal(20,2),
					@NewValue decimal(20,2)
			SET @OldQuantity = (SELECT Quantity FROM OrderDetails
							WHERE OrderID = @OrderID
AND DishID = @DishID)
			SET @DishPrice = (SELECT UnitPrice FROM OrderDetails
							WHERE OrderID = @OrderID
AND DishID = @DishID)
			SET @OrderValue = dbo.GetOrderValue(@OrderID)
			UPDATE OrderDetails
			SET Quantity = @NewQuantity
			WHERE OrderID = @OrderID AND DishID = @DishID

			SET @NewValue = @OrderValue +
(@NewQuantity-@OldQuantity)*@DishPrice

			EXEC dbo.UpdateValue @OrderID, @NewValue

		COMMIT TRAN TUpdateDishQuantity
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TUpdateDishQuantity
		DECLARE @msg nvarchar(2048) =
		'Blad zmiany ilosci dania w
```

```
zamowieniu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddRefund

Dodaje zwrot

```
CREATE PROCEDURE [dbo].[AddRefund]
      @ReceiptID int,
      @OrderID int,
      @Value decimal(20,2) = 0,
      @Refunded bit = 0

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TAddRefund
                  INSERT INTO Refunds(ReceiptID, OrderID, Value, Refunded)
                  VALUES (@ReceiptID, @OrderID, @Value, @Refunded)
            COMMIT TRAN TAddRefund
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TAddRefund
            DECLARE @msg nvarchar(2048) =
            'Blad dodawania zwrotu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

UpdateStatus

Zmienia status zwrotu

```
CREATE PROCEDURE [dbo].[UpdateStatus]
      @ReceiptID int,
      @OrderID int,
      @Refunded bit = 1

AS
BEGIN
      SET NOCOUNT ON;
      BEGIN TRY
            BEGIN TRAN TUpdateStatus
                  UPDATE Refunds
                  SET Refunded = @Refunded
```

```
                  WHERE ReceiptID = @ReceiptID AND OrderID = @OrderID
            COMMIT TRAN TUpdateStatus
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TUpdateStatus
            DECLARE @msg nvarchar(2048) =
            'Blad zmiany statusu
zwrotu:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

AddCollectiveInvoice

Dodaje rachunek zbiorowy

```
CREATE PROCEDURE [dbo].[AddCollectiveInvoice]
      @ReceiptID int,
      @ValidFrom datetime,
      @ValidTo datetime,
      @InProgress bit = 1

AS
BEGIN
      SET NOCOUNT ON
      BEGIN TRY
            BEGIN TRAN TAddCollectiveInvoice
                  INSERT INTO CollectiveInvoices(ReceiptID, ValidFrom,
ValidTo, InProgress)
                  VALUES (@ReceiptID, @ValidFrom, @ValidTo, @InProgress)
            COMMIT TRAN TAddCollectiveInvoice
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TAddCollectiveInvoice
            DECLARE @msg nvarchar(2048) =
            'Blad dodawania rachunku
zbiorczego:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

UpdateCollectiveInvoiceStatus

Zmienia status rachunku zbiorowego - z otwartego na zamknięty

```
CREATE PROCEDURE [dbo].[UpdateCollectiveInvoiceStatus]
      @ReceiptID int,
      @InProgress bit = 0
```

```
AS
BEGIN
	SET NOCOUNT ON
	BEGIN TRY
		BEGIN TRAN TUpdateCollectiveInvoiceStatus
			UPDATE CollectiveInvoices
			SET InProgress = @InProgress
			WHERE ReceiptID = @ReceiptID
			DECLARE @date datetime
			SET @date = GETDATE()
			EXEC dbo.InvoiceReceipt @ReceiptID, @date;
		COMMIT TRAN TAddCollectiveInvoice
	END TRY
	BEGIN CATCH
		ROLLBACK TRAN TUpdateCollectiveInvoiceStatus
		DECLARE @msg nvarchar(2048) =
		'Blad dodawania rachunku
zbiorczego:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
		THROW 52000, @msg, 1;
	END CATCH
END
GO
```

CheckCollectiveInvoices

Sprawdza, czy należy zamknąć rozliczanie jakiejś faktury zbiorczej

```
CREATE PROCEDURE [dbo].[CheckCollectiveInvoices]
AS
BEGIN
	SET NOCOUNT ON
	WAITFOR TIME '00:00';
	BEGIN TRY
		BEGIN TRAN TCheckCollectiveInvoices
			DECLARE @MyCursor CURSOR
			DECLARE @ReceiptID int, @ValidTo datetime
			SET @MyCursor = CURSOR FOR
			SELECT ReceiptID, ValidTo FROM CollectiveInvoices WHERE
InProgress = 1

			OPEN @MyCursor
			FETCH NEXT FROM @MyCursor
			INTO @ReceiptID, @ValidTo

			WHILE @@FETCH_STATUS = 0
			BEGIN
				IF GETDATE() > @ValidTo
				BEGIN
					DECLARE @date datetime
					SET @date = GETDATE()
```

```
                                EXEC dbo.UpdateCollectiveInvoiceStatus
@ReceiptID, 0
                                EXEC dbo.InvoiceReceipt @ReceiptID, @date
                          END
                          FETCH NEXT FROM @MyCursor
                          INTO @ReceiptID, @ValidTo
                    END;

                    CLOSE @MyCursor ;
                    DEALLOCATE @MyCursor;
              COMMIT TRAN TCheckCollectiveInvoices
      END TRY
      BEGIN CATCH
            ROLLBACK TRAN TCheckCollectiveInvoices
            DECLARE @msg nvarchar(2048) =
            'Blad sprawdzania czy faktura zbiorcza powinna byc
zamknieta:'+CHAR(13)+CHAR(10)+ERROR_MESSAGE();
            THROW 52000, @msg, 1;
      END CATCH
END
GO
```

4.Funkcje

Spis funkcji:

1. CheckIfIndividual
2. GetSixthDiscount
3. CheckReservationConditions
4. GetCategoryID
5. GetCurrentMenu
6. GetUpcomingMenu
7. GetMenuInsertionDate
8. GetCurrentDishPrice
9. GetDishRemovalDate
10. GetNumberOfDishesInMenu
11. GetNumberOfSimilarDishes
12. CanAddDishToMenu
13. DoesDishIncludeCategory
14. GetReceiptValue
15. GetValueAfterDiscount
16. GetOrdersFromReceipt
17. GetOrderReceipt
18. IsSettled
19. GetOrderValue
20. IsReceived
21. GetBookedDate
22. GetOrderDate
23. GetCustomerFromOrder
24. GetOrderDiscount
25. CanAddDishToOrder
26. IsCollectiveInvoice
27. GetNumberOfFreeSeats
28. OrdersPerClient
29. DiscountsPerClient

CheckIfIndividual

Funkcja zwracająca prawdę, gdy podany klient jest indywidualny, lub fałsz, gdy jest firmą.

```
CREATE FUNCTION CheckIfIndividual (@CustomerID int)
RETURNS BIT
AS
BEGIN
    DECLARE @result bit
```

```
    IF ((SELECT CustomerType FROM Customers WHERE CustomerID = @CustomerID)
= 'individual')
        SET @result = 1
    ELSE SET @result = 0

    RETURN @result

END
GO
```

GetSixthDiscount

Funkcja zwracająca wartość aktywnej szóstej zniżki(kwotowej) dla danego klienta.

```
CREATE FUNCTION GetSixthDiscount (@CustomerID int)
RETURNS TABLE
AS
RETURN
(
    SELECT *
    FROM GrantedDiscounts gd
    WHERE gd.CustomerID = @CustomerID AND gd.DiscountID = 6 AND gd.Used
= 0
);
GO
```

GetNumberOfFreeSeats

Funkcja zwraca ilość wszystkich dostępnych miejsc.

```
CREATE FUNCTION GetNumberOfFreeSeats()
RETURNS int
AS
BEGIN
      RETURN (SELECT SUM(NumberOfSeats) FROM Tables WHERE Available =
TRUE)
END
GO
```

CheckReservationConditions

Funkcja sprawdzająca, czy dany klient może złożyć rezerwację miejsca na podstawie złożonego zamówienia. Sprawdzana jest odpowiednia ilość wolnych miejsc w placówce oraz zgodność kwoty za zamówienie z warunkami, jakie klient musi spełnić.

```
CREATE FUNCTION CheckReservationConditions (@CustomerID int, @OrderID
int, @NumberOfPeople int)
RETURNS bit
```

```
AS
BEGIN
    DECLARE @result bit, @isIndividual bit, @AvailablePlaces int

    EXEC @AvailablePlaces = GetNumberOfFreeSeats

    EXEC @isIndividual = CheckIfIndividual @CustomerID

    IF (@CustomerID IS NULL OR @OrderID IS NULL OR @AvailablePlaces <
@NumberOfPeaple)
        SET @result = 0
    ELSE IF @isIndividual = 0
        SET @result = 1
    ELSE
        BEGIN
            DECLARE @OrderValue int, @NumberOfCustomerOrders int
            SELECT @OrderValue = Orders.Value FROM Orders WHERE OrderID
= @OrderID
            SELECT @NumberOfCustomerOrders = COUNT(*) FROM Orders WHERE
CustomerID = @CustomerID

            IF (@OrderValue >= 200 OR (@OrderValue >=50 AND
@NumberOfCustomerOrders >= 5))
                SET @result = 1
            ELSE SET @result = 0
        END

    RETURN @result

END
GO
```

GetCategoryID

Funkcja zwraca ID kategorii na podstawie podanej nazwy.

```
CREATE FUNCTION GetCategoryID(@CategoryName nvarchar(30))
RETURNS int
AS
BEGIN
      RETURN (SELECT CategoryID FROM Categories WHERE CategoryName =
@CategoryName)
END
GO
```

GetCurrentMenu

Funkcja zwraca obecnie obowiązujące menu.

```
CREATE FUNCTION GetCurrentMenu()
RETURNS int
AS
BEGIN
	RETURN (SELECT MenuID FROM Menu WHERE InsertionDate <= GETDATE() AND
RemovalDate >= GETDATE());
END
GO
```

GetUpcomingMenu

Funkcja zwraca menu, które zostało zaplanowane, ale jeszcze nie obowiązuje w restauracji.

```
CREATE FUNCTION GetUpcomingMenu()
RETURNS int
AS
BEGIN
	RETURN (SELECT TOP(1) MenuID FROM Menu WHERE InsertionDate > GETDATE()
AND RemovalDate >= GETDATE());
END
GO
```

GetMenuInsertionDate

Funkcja zwraca datę wprowadzenia menu o podanym ID.

```
CREATE FUNCTION GetMenuInsertionDate(@MenuID int)
RETURNS datetime
AS
BEGIN
	RETURN (SELECT InsertionDate FROM Menu WHERE MenuID = @MenuID)
END
GO
```

GetNumberOfDishesInMenu

Funkcja zwraca ilość dań, znajdujących się w menu o podanym ID.

```
CREATE FUNCTION GetNumberOfDishesInMenu(@MenuID int)
RETURNS int
AS
BEGIN
	RETURN (SELECT COUNT(*) FROM MenuDetails WHERE MenuID = @MenuID);
END
GO
```

GetNumberOfSimilarDishes

Funkcja przyjmuje parę menu, a następnie oblicza ile dań powtarza się w tych menu.

```
CREATE FUNCTION GetNumberOfSimilarDishes(@Menu1 int, @Menu2 int)
RETURNS int
AS
BEGIN
	DECLARE @result int
	SET @result = (SELECT COUNT (*)
			FROM MenuDetails MD1
			CROSS JOIN MenuDetails MD2
			WHERE MD1.MenuID = @Menu1 AND
    MD2.MenuID = @Menu2 AND MD1.DishID = MD2.DishID);
	RETURN @result
END
GO
```

CanAddDishToMenu

Funkcja sprawdza, czy danie może zostać dodane do menu. Sprawdzane są odpowiednie kryteria, wynikające ze specyfikacji projektu. Wspomniane kryteria to: czy ostatnia data usunięcia dania z menu to przynajmniej miesiąc od chwili, w której chcemy dodać danie oraz jeśli danie było w poprzednim menu, to czy po dodaniu dania nie zostanie przekroczona dopuszczalna ilość powtarzających się dań.

```
CREATE FUNCTION CanAddDishToMenu(@DishID int, @MenuID int)
RETURNS bit
AS
BEGIN
	DECLARE @CurrentMenu int, @RemovalDate datetime, @InsertionDate
datetime
	EXEC @InsertionDate = dbo.GetMenuInsertionDate @MenuID --Data w ktorej
planujemy wprowadzic nowe menu
	EXEC @CurrentMenu = dbo.GetCurrentMenu --Aktualnie obowiazujace menu
	EXEC @RemovalDate = dbo.GetDishRemovalDate @DishID --Data wycofania
dania z jakiegos menu
	IF (DATEDIFF(MONTH, @RemovalDate, @InsertionDate) < 1)
	BEGIN
		RETURN 0;
	END
	ELSE IF ((SELECT COUNT(*) FROM MenuDetails WHERE MenuID = @CurrentMenu
AND DishID = @DishID) != 0)
	--Danie bylo w poprzednim menu
	BEGIN
		DECLARE @SimilarDishes int, @NumberOfDIshes int
		EXEC @SimilarDishes = dbo.GetNumberOfSimilarDishes
@CurrentMenu, @MenuID
		EXEC @NumberOfDIshes = dbo.GetNumberOfDishesInMenu @CurrentMenu
		IF ((@SimilarDishes+1) > @NumberOfDIshes/2)
		BEGIN
			RETURN 0;
		END
	END
```

```
	RETURN 1;
END
GO
```

GetCurrentDishPrice

Funkcja zwraca obecnie obowiązującą cenę za danie.

```
CREATE FUNCTION GetCurrentDishPrice(@DishID int)
RETURNS decimal(20,2)
AS
BEGIN
	DECLARE @Result decimal(20,2) = (SELECT DishPrice FROM Dishes WHERE
DishID = @DishID)
	RETURN @Result
END
GO
```

GetDishRemovalDate

Funkcja zwraca datę ostatniego usuniecia dania z jakiegoś menu. Jeśli danie jeszcze nigdy nie zostało usuniete z menu, zwracany jest NULL.

```
CREATE FUNCTION GetDishRemovalDate(@DishID int)
RETURNS datetime
AS
BEGIN
	RETURN (SELECT RemovalDate FROM Dishes WHERE DishID = @DishID)
END
GO
```

DoesDishIncludeCategory

Funkcja sprawdza, czy danie zawiera produkt danej kategorii, jeśli tak to zwracana jest prawda, w przeciwnym wypadku fałsz.

```
CREATE FUNCTION DoesDishIncludeCategory(@DishID int, @CategoryID int)
RETURNS bit
AS
BEGIN
	IF (SELECT COUNT(*) FROM DishDetails DD
		INNER JOIN Products P
		ON DD.ProductID = P.ProductID
		WHERE DishID = @DishID AND P.CategoryID = @CategoryID) > 0
	BEGIN
		RETURN 1;
	END
	RETURN 0;
END
GO
```

GetReceiptValue

Funkcja zwraca wartość rachunku.

```
CREATE FUNCTION GetReceiptValue(@ReceiptID int)
RETURNS decimal(20,2)
AS
BEGIN
	RETURN (SELECT Value FROM Receipt WHERE ReceiptID = @ReceiptID);
END
GO
```

GetOrdersFromReceipt

Funkcja zwraca listę zamówień, wchodzących w skład danego racunku.

```
CREATE FUNCTION GetOrdersFromReceipt(@ReceiptID int)
RETURNS TABLE
AS
	RETURN (SELECT OrderID FROM ReceiptDetails WHERE ReceiptID =
@ReceiptID);
GO
```

GetOrderReceipt

Funkcja zwraca ID rachunku, do którego należy dane zamówienie.

```
CREATE FUNCTION GetOrderReceipt(@OrderID int)
RETURNS int
AS
BEGIN
	RETURN (SELECT ReceiptID FROM ReceiptDetails WHERE OrderID =
@OrderID);
END
GO
```

IsSettled

Funkcja sprawdza, czy zamówienie zostało opłacone i jeśli zostało, to zwraca prawdę, w przeciwnym wypadku fałsz.

```
CREATE FUNCTION IsSettled(@OrderID int)
RETURNS bit
AS
BEGIN
	RETURN (SELECT Settled FROM Orders WHERE OrderID = @OrderID)
END
```

```
GO
```

GetOrderValue

Funkcja zwraca aktualną wartość zamówienia.

```
CREATE FUNCTION GetOrderValue(@OrderID int)
RETURNS bit
AS
BEGIN
	RETURN (SELECT Value FROM Orders WHERE OrderID = @OrderID)
END
GO
```

IsReceived

Funkcja sprawdza, czy zamówienie zostało odebrane i zwraca odpowienio prawdę lub fałsz.

```
CREATE FUNCTION IsReceived(@OrderID int)
RETURNS bit
AS
BEGIN
	IF ((SELECT Received FROM Orders WHERE OrderID = @OrderID) IS NULL OR
(SELECT Received FROM Orders WHERE OrderID = @OrderID) = 0)
	BEGIN
	RETURN 0;
	END
	RETURN 1;
END
GO
```

GetBookedDate

Funkcja zwraca datę, na którą zostało dokonane zamówienie.

```
CREATE FUNCTION GetBookedDate(@OrderID int)
RETURNS datetime
AS
BEGIN
	RETURN(SELECT ReceiveDate FROM Orders WHERE OrderID = @OrderID)
END
GO
```

GetOrderDate

Funkcja zwraca datę, w której złożono zamówienie.

```
CREATE FUNCTION GetOrderDate(@OrderID int)
RETURNS datetime
AS
```

```
BEGIN
	RETURN(SELECT OrderDate FROM Orders WHERE OrderID = @OrderID)
END
GO
```

GetCustomerFromOrder

Funkcja zwraca ID klienta, który złożył zamówienie.

```
CREATE FUNCTION GetCustomerFromOrder(@OrderID int)
RETURNS int
AS
BEGIN
	RETURN(SELECT CustomerID FROM Orders WHERE OrderID = @OrderID)
END
GO
```

GetOrderDiscount

Funkcja zwraca kwotę zniżki, która została przyznana zamówieniu.

```
CREATE FUNCTION GetOrderDiscount(@OrderID int)
RETURNS decimal(20,2)
AS
BEGIN
	RETURN(SELECT Discount FROM Orders WHERE OrderID = @OrderID)
END
GO
```

GetValueAfterDiscount

Funkcja zwraca wartość zamówienia, po uwzględnieniu zniżki.

```
CREATE FUNCTION GetValueAfterDiscount(@OrderID int)
RETURNS decimal(20,2)
AS
BEGIN
	RETURN(SELECT Value-Discount FROM Orders WHERE OrderID = @OrderID)
END
GO
```

CanAddDishToOrder

Funkcja sprawdza, czy danie może zostać dodane do zamówienia. W ogólnym przypadku sprawdzane jest, czy danie jest aktualnie dostępne w menu, natomiast w przypadku dań z kategorii seafood sprawdzane są kryteria opisane w specyfikacji projektu.

```
CREATE FUNCTION CanAddDishToOrder(@DishID int, @OrderID int)
RETURNS bit
AS
BEGIN
	DECLARE @CurrentMenu int, @SeafoodID int, @DoesIncludeSeafood bit,
```

```
@BookedDate datetime, @OrderDate datetime
	EXEC @CurrentMenu = dbo.GetCurrentMenu --Aktualnie obowiazujace menu
	EXEC @SeafoodID = dbo.GetCategoryID 'seafood'
	EXEC @DoesIncludeSeafood = dbo.DoesDishIncludeCategory @DishID,
@SeafoodID
	EXEC @BookedDate = dbo.GetBookedDate @OrderID
	EXEC @OrderDate = dbo.GetOrderDate @OrderID
	IF (@DoesIncludeSeafood = 1)
	BEGIN
		IF (DATENAME(DW, @BookedDate) IN ('Thursday', 'Friday',
'Saturday') AND DATEDIFF(DAY, @OrderDate, @BookedDate) >= DATEPART(DW,
@BookedDate)-1)
		BEGIN
			RETURN 1;
		END
	END
	ELSE IF ((SELECT Availability FROM MenuDetails WHERE MenuID =
@CurrentMenu AND DishID = @DishID) = 1)
	--Danie jest dostepne
	BEGIN
		RETURN 1;
	END
	RETURN 0;
END
GO
```

IsCollectiveInvoice

Funkcja na podstawie ID rachunku sprawdza, czy rachunek ten jest fakturą zbiorczą.

```
CREATE FUNCTION IsCollectiveInvoice(@ReceiptID int)
RETURNS bit
AS
BEGIN
	IF ((SELECT COUNT(*) FROM CollectiveInvoices WHERE ReceiptID =
@ReceiptID) = 1)
	BEGIN
		RETURN 1;
	END
	RETURN 0;
END
GO
```

OrdersPerClient

Funkcja listuje wszystkie zamówienia złożóne przez danego klienta.

```
CREATE FUNCTION [dbo].[OrdersPerClient] (@CustomerID int)
RETURNS TABLE
AS
```

```
RETURN (SELECT *
            FROM Orders
            WHERE CustomerID = @CustomerID);
GO
```

DiscountsPerClient

Funkcja listuje wszystkie rabaty, które zostały kiedykolwiek przyznane danemu klientowi.

```
CREATE FUNCTION [dbo].[DiscountsPerClient] (@CustomerID int)
RETURNS TABLE
AS
RETURN (SELECT *
            FROM GrantedDiscounts
            WHERE CustomerID = @CustomerID);
GO
```

5.Trigger

Trigger OrderUpdated uruchamia się w momencie aktualizacji danych zamówienia. Następnie sprawdza, czy zamówienie zostało zrealizowane (Received) oraz opłacone (Settled). Gdy ten warunek zachodzi, trigger uruchamia procedurę UpdateDataForDiscounts aktualizującą statystyki klienta opłacającego zamówienie.

```
CREATE TRIGGER OrderUpdated
      ON dbo.Orders
AFTER UPDATE
AS
BEGIN
      SET NOCOUNT ON;

      --tu wyciągane jest zamówienie, które uruchomiło trigger
      DECLARE @OrderID int
      SELECT
      @OrderID = inserted.OrderID
      FROM
      inserted
      INNER JOIN
      deleted
      ON inserted.OrderID = deleted.OrderID

    DECLARE @ReceiptID int
    EXEC @ReceiptID = GetOrderReceipt @OrderID

      DECLARE @CustomerID int
      SELECT @CustomerID = InvoicedCustomer FROM Receipt WHERE ReceiptID
= @ReceiptID

      DECLARE @OrderValue DECIMAL(20,2)
```

```
      SELECT @OrderValue = Value FROM Receipt WHERE ReceiptID =
@ReceiptID

      --jeśli po update settled i received są na 1 uruchamiam procedurę
aktulizacji danych
      IF ((SELECT Settled FROM Orders WHERE OrderID = @OrderID) = 1 AND
(SELECT Received FROM Orders WHERE OrderID = @OrderID) = 1)
      EXEC UpdateDataForDiscounts
            @CustomerID,
            @OrderValue

END
GO
```

6.Indeksy

Tabela Products: CategoryID

```
CREATE NONCLUSTERED INDEX [Products_CategoryID_Index] ON
[dbo].[Products]
(
[CategoryID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
```

Tabela Orders: CustomerID, EmployeeID

```
CREATE NONCLUSTERED INDEX [Orders_CustomerID_Index] ON [dbo].[Orders]
(
[CustomerID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)

CREATE NONCLUSTERED INDEX [Orders_EmployeeID_Index] ON [dbo].[Orders]
(
[EmployeeID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
```

Tabela IndividualCustomers: CompanyID

```
CREATE NONCLUSTERED INDEX [IndividualCustomers_CompanyID_Index] ON
[dbo].[IndividualCustomers]
(
[CompanyID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
```

Tabela GrantedDiscounts: CustomerID, DiscountID

```
CREATE NONCLUSTERED INDEX [GrantedDiscounts_CustomerID_Index] ON
[dbo].[GrantedDiscounts]
(
[CustomerID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)

CREATE NONCLUSTERED INDEX [GrantedDiscounts_DiscountID_Index] ON
[dbo].[GrantedDiscounts]
(
[DiscountID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
```

Tabela Reservations: OrderID, CustomerID

```
CREATE UNIQUE NONCLUSTERED INDEX [Reservations_OrderID_Index] ON
[dbo].[Reservations]
(
[OrderID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)

CREATE NONCLUSTERED INDEX [Reservations_CustomerID_Index] ON
[dbo].[Reservations]
(
[CustomerID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
```

Tabela Receipt: InvoicedCustomer (CustomerID)

```
CREATE NONCLUSTERED INDEX [Receipt_InvoicedCustomer_Index] ON
[dbo].[Receipt]
(
[InvoicedCustomer] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB =
OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
```

7.Tworzenie raportów (widoki)

Tworzenie raportów dotyczących zamówień, rabatów itd. zrealizowaliśmy za pomocą widoków. Każdy widok zwraca odpowiednią tabelę z określonego czasu, zawierającą określone dane. Do generowania raportów służą również dwie funkcje: OrdersPerClient oraz DiscountsPerClient.

Spis widoków:

1. LastWeekReservations
2. LastMonthReservations
3. AllReservationsCount
4. LastWeekIndividualCustomerDiscounts
5. LastMonthIndividualCustomerDiscounts
6. LastWeekCompanyDiscount
7. LastMonthCompanyDiscount
8. ActiveCustomerDiscountCount
9. CurrentMenu
10. AverageOrderPriceInd
11. AverageOrderPriceComp
12. AverageOrderPriceIndWeek
13. AverageOrderPriceCompWeek
14. SumOrderPriceIndWeek
15. SumOrderPriceCompWeek

LastWeekReservations

Widok zwraca wszystkie rezerwacje z poprzedniego tygodnia.

```
CREATE VIEW [dbo].[LastWeekReservations] AS
SELECT dbo.Reservations.ReservationID,
dbo.Customers.CustomerID,
dbo.Reservations.NumberOfPeople,
dbo.Orders.OrderDate,
dbo.Orders.ReceiveDate
FROM dbo.Reservations
INNER JOIN dbo.Customers
ON dbo.Reservations.CustomerID = dbo.Customers.CustomerID
INNER JOIN dbo.Orders
ON dbo.Orders.OrderID = dbo.Reservations.OrderID
WHERE (DATEPART(week, dbo.Orders.OrderDate) = DATEPART(week,
GETDATE())-1 AND YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.Orders.OrderDate) = 53 AND YEAR(dbo.Orders.OrderDate) =
```

```
YEAR(GETDATE()) -1)
GO
```

LastMonthReservations

Widok zwraca wszystkie rezerwacje z poprzedniego miesiąca.

```
CREATE VIEW [dbo].[LastMonthReservations] AS
SELECT dbo.Reservations.ReservationID,
dbo.Customers.CustomerID,
dbo.Reservations.NumberOfPeople,
dbo.Orders.OrderDate,
dbo.Orders.ReceiveDate
FROM dbo.Reservations
INNER JOIN dbo.Customers
ON dbo.Reservations.CustomerID = dbo.Customers.CustomerID
INNER JOIN dbo.Orders
ON dbo.Orders.OrderID = dbo.Reservations.OrderID
WHERE (MONTH(dbo.Orders.OrderDate) = MONTH(GETDATE())-1 AND
YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()))
OR (MONTH(GETDATE()) = 1 AND MONTH(dbo.Orders.OrderDate) = 12 AND
YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()) -1)
GO
```

AllReservationCount

Widok zwraca ilość rezerwacji dla poszczególnych klientów.

```
CREATE VIEW [dbo].[AllReservationsCount] AS
SELECT dbo.Reservations.CustomerID,
COUNT(dbo.Reservations.ReservationID) AS AmountOfReservations
FROM dbo.Reservations
GROUP BY dbo.Reservations.CustomerID
GO
```

LastWeekIndividualCustomerDiscounts

Widok zwraca rabaty klientów indywidualnych z zeszłego tygodnia.

```
CREATE VIEW [dbo].[LastWeekIndividualCustomerDiscounts] AS
SELECT dbo.IndividualCustomers.IndividualCustomerID,
dbo.GrantedDiscounts.DiscountID,
dbo.GrantedDiscounts.DiscountValue,
dbo.GrantedDiscounts.GrantedDate,
dbo.GrantedDiscounts.ExpirationDate,
dbo.GrantedDiscounts.Used
FROM dbo.IndividualCustomers
LEFT JOIN dbo.GrantedDiscounts
ON dbo.IndividualCustomers.IndividualCustomerID =
```

```
dbo.GrantedDiscounts.CustomerID
WHERE (DATEPART(week, dbo.GrantedDiscounts.GrantedDate) = DATEPART(week,
GETDATE())-1 AND YEAR(dbo.GrantedDiscounts.GrantedDate) =
YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.GrantedDiscounts.GrantedDate) = 53 AND
YEAR(dbo.GrantedDiscounts.GrantedDate) = YEAR(GETDATE()) -1)
GO
```

LastMonthIndividualCustomerDiscounts

Widok zwraca rabaty klientów indywidualnych z zeszłego miesiąca.

```
CREATE VIEW [dbo].[LastMonthIndividualCustomerDiscounts] AS
SELECT dbo.IndividualCustomers.IndividualCustomerID,
dbo.GrantedDiscounts.DiscountID,
dbo.GrantedDiscounts.DiscountValue,
dbo.GrantedDiscounts.GrantedDate,
dbo.GrantedDiscounts.ExpirationDate,
dbo.GrantedDiscounts.Used
FROM dbo.IndividualCustomers
LEFT JOIN dbo.GrantedDiscounts
ON dbo.IndividualCustomers.IndividualCustomerID =
dbo.GrantedDiscounts.CustomerID
WHERE (MONTH(dbo.GrantedDiscounts.GrantedDate) = MONTH(GETDATE())-1 AND
YEAR(dbo.GrantedDiscounts.GrantedDate) = YEAR(GETDATE()))
OR (MONTH(GETDATE()) = 1 AND MONTH(dbo.GrantedDiscounts.GrantedDate) =
12 AND YEAR(dbo.GrantedDiscounts.GrantedDate) = YEAR(GETDATE()) -1)
GO
```

LastWeekCompanyDiscounts

Widok zwraca rabaty firm z zeszłego tygodnia.

```
CREATE VIEW [dbo].[LastWeekCompanyDiscounts] AS
SELECT dbo.Companies.CompanyID,
dbo.GrantedDiscounts.DiscountID,
dbo.GrantedDiscounts.DiscountValue,
dbo.GrantedDiscounts.GrantedDate,
dbo.GrantedDiscounts.ExpirationDate,
dbo.GrantedDiscounts.Used
FROM dbo.Companies
LEFT JOIN dbo.GrantedDiscounts
ON dbo.Companies.CompanyID = dbo.GrantedDiscounts.CustomerID
WHERE (DATEPART(week, dbo.GrantedDiscounts.GrantedDate) = DATEPART(week,
GETDATE())-1 AND YEAR(dbo.GrantedDiscounts.GrantedDate) =
YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.GrantedDiscounts.GrantedDate) = 53 AND
```

```
YEAR(dbo.GrantedDiscounts.GrantedDate) = YEAR(GETDATE()) -1)
GO
```

LastMonthCompanyDiscounts

Widok zwraca rabaty firm z zeszłego miesiąca.

```
CREATE VIEW [dbo].[LastMonthCompanyDiscounts] AS
SELECT dbo.Companies.CompanyID,
dbo.GrantedDiscounts.DiscountID,
dbo.GrantedDiscounts.DiscountValue,
dbo.GrantedDiscounts.GrantedDate,
dbo.GrantedDiscounts.ExpirationDate,
dbo.GrantedDiscounts.Used
FROM dbo.Companies
LEFT JOIN dbo.GrantedDiscounts
ON dbo.Companies.CompanyID = dbo.GrantedDiscounts.CustomerID
WHERE (MONTH(dbo.GrantedDiscounts.GrantedDate) = MONTH(GETDATE())-1 AND
YEAR(dbo.GrantedDiscounts.GrantedDate) = YEAR(GETDATE()))
OR (MONTH(GETDATE()) = 1 AND MONTH(dbo.GrantedDiscounts.GrantedDate) =
12 AND YEAR(dbo.GrantedDiscounts.GrantedDate) = YEAR(GETDATE()) -1)
GO
```

ActiveCustomerDiscountCount

Widok zwraca ilość aktywnych zniżek dla poszczególnych klientów.

```
CREATE VIEW [dbo].[ActiveCustomerDiscountCount] AS
SELECT dbo.GrantedDiscounts.CustomerID,
COUNT(dbo.GrantedDiscounts.GrantedDiscountID) AS AmountOfDiscounts
FROM dbo.GrantedDiscounts
WHERE dbo.GrantedDiscounts.Used = 0
GROUP BY dbo.GrantedDiscounts.CustomerID
GO
```

CurrentMenu

Widok zwraca aktualnie obowiązujące menu.

```
CREATE VIEW [dbo].[CurrentMenu] AS
SELECT D.DishName, MD.DishPrice, MD.Availability
FROM dbo.MenuDetails AS MD
INNER JOIN Dishes AS D
ON MD.DishID = D.DishID
WHERE MD.MenuID = dbo.GetCurrentMenu()
GO
```

AverageOrderPriceInd

Widok zwraca średnią wartość z zamówień klientów indywidulanych.

```
CREATE VIEW [dbo].[AverageOrderPriceInd] AS
SELECT ROUND(AVG(Value),2) AS 'Average price'
FROM Orders
WHERE dbo.CheckIfIndividual(CustomerID) = 1
GO
```

AverageOrderPriceComp

Widok zwraca średnią wartość z zamówień firm.

```
CREATE VIEW [dbo].[AverageOrderPriceComp] AS
SELECT ROUND(AVG(Value),2) AS 'Average price'
FROM Orders
WHERE dbo.CheckIfIndividual(CustomerID) = 0
GO
```

AverageOrderPriceIndWeek

Widok zwraca średnią wartość z zamówień klientów indywidulanych z zeszłego tygodnia.

```
CREATE VIEW [dbo].[AverageOrderPriceIndWeek] AS
SELECT ROUND(AVG(Value),2) AS 'Average price'
FROM Orders
WHERE dbo.CheckIfIndividual(CustomerID) = 1
AND ((DATEPART(week, dbo.Orders.OrderDate) = DATEPART(week, GETDATE())-1
AND YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.Orders.OrderDate) = 53 AND YEAR(dbo.Orders.OrderDate) =
YEAR(GETDATE()) -1))
GO
```

AverageOrderPriceCompWeek

Widok zwraca średnią wartość z zamówień firm z zeszłego tygodnia.

```
CREATE VIEW [dbo].[AverageOrderPriceCompWeek] AS
SELECT ROUND(AVG(Value),2) AS 'Average price'
FROM Orders
WHERE dbo.CheckIfIndividual(CustomerID) = 0
AND ((DATEPART(week, dbo.Orders.OrderDate) = DATEPART(week, GETDATE())-1
AND YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.Orders.OrderDate) = 53 AND YEAR(dbo.Orders.OrderDate) =
YEAR(GETDATE()) -1))
GO
```

SumOrderPriceIndWeek

Widok zwraca sumę wartości zamówień klientów indywidulanych z zeszłego tygodnia.

```
CREATE VIEW [dbo].[SumOrderPriceIndWeek] AS
SELECT SUM(Value) AS 'Sum price'
```

```
FROM Orders
WHERE dbo.CheckIfIndividual(CustomerID) = 1
AND ((DATEPART(week, dbo.Orders.OrderDate) = DATEPART(week, GETDATE())-1
AND YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.Orders.OrderDate) = 53 AND YEAR(dbo.Orders.OrderDate) =
YEAR(GETDATE()) -1))
GO
```

SumOrderPriceCompWeek

Widok zwraca sumę wartości zamówień firm z zeszłego tygodnia.

```
CREATE VIEW [dbo].[SumOrderPriceCompWeek] AS
SELECT SUM(Value) AS 'Sum price'
FROM Orders
WHERE dbo.CheckIfIndividual(CustomerID) = 0
AND ((DATEPART(week, dbo.Orders.OrderDate) = DATEPART(week, GETDATE())-1
AND YEAR(dbo.Orders.OrderDate) = YEAR(GETDATE()))
OR (DATEPART(week, GETDATE()) = 1 AND DATEPART(week,
dbo.Orders.OrderDate) = 53 AND YEAR(dbo.Orders.OrderDate) =
YEAR(GETDATE()) -1))
GO
```

8.Generowanie danych.

By wygenerować przykładowe dane, posłużyliśmy się narzędziem SQL Data Generator 4 firmy RedGate. Wygenerowane dane spełniają wszystkie warunki integralnościowe.

9.Test poprawności działania wybranych elementów bazy danych

Dodajemy menu, które będzie obowiązywało od GETDATE(), aż do GETDATE() + 2 tygodnie. W tym celu chwilowo modyfikujemy procedurę AddMenu tak, aby nie sprawdzała czy dodajemy menu z co najmniej jednodniowym wyprzedzeniem.

```
DECLARE @InsertionDate DATETIME = GETDATE()
EXEC dbo.AddMenu @InsertionDate
```

Następnie dodajemy dania do menu. Dla przykładu zostało dodane 7 dań.

```
DECLARE @CurrentMenu int = dbo.GetCurrentMenu()
EXEC dbo.AddDishToMenu @CurrentMenu, 100
EXEC dbo.AddDishToMenu @CurrentMenu, 200
EXEC dbo.AddDishToMenu @CurrentMenu, 300
EXEC dbo.AddDishToMenu @CurrentMenu, 400
EXEC dbo.AddDishToMenu @CurrentMenu, 500
EXEC dbo.AddDishToMenu @CurrentMenu, 600
EXEC dbo.AddDishToMenu @CurrentMenu, 700
```

Zawartość menu sprawdzamy w MenuDetails (ID dodanego menu to w przykładzie 1001).

	MenuID	DishID	DishPrice	Availability
1	1001	100	41.92	1
2	1001	200	55.57	1
3	1001	300	30.78	1
4	1001	400	3.67	1
5	1001	500	42.27	1
6	1001	600	71.32	1
7	1001	700	28.06	1

Następnie dodajemy kilku przykładowych klientów, jeden w pełni indywidualny, jeden indywidualny-firmowy, jeden firmowy rozliczający się przez fakturę zbiorczą i jeden zwykły klient firmowy.

```
EXEC dbo.AddCompany 'BazoDaneX', '1234567890', 'Poland', '12-345', 'ul.
SQLA 128', 'Bajtocja', -1, 0, 0, 0, 0, '123456789',
'BazoDaneX@contact.com', NULL
EXEC dbo.AddCompany 'BazoDaneY', '0987654321', 'Poland', '12-345', 'ul.
SQLA 64', 'Bajtocja', 1, 0, 0, 0, 0, '987654321',
'BazoDaneY@contact.com', NULL
EXEC dbo.AddIndividualCustomer NULL, 'Marcin', 'Najman', 41, 1, 0, 0, 0,
'individual', '913218321', 'najman@conact.com', NULL
EXEC dbo.AddIndividualCustomer 1001, 'Bill', 'Gates', 65, 1, 0, 0, 0,
'individual', '141535141', 'gates@conact.com', NULL
```

Następnie symulujemy składanie zamówienia dla przykładowego klienta.

Sprawdzamy aktualne informacje o kliencie w bazie.

	IndividualCustomerID	CompanyID	FirstName	LastName	Age	PersonalDataAgreement	NumberOfOrders	OrdersInRow	SumOfOrders
1	1004	NULL	Marcin	Najman	41	1	2	2	85.27

Dodajemy zamówienie. Na potrzeby przykładu dodaliśmy zamówienie, które klient odbierze za 15 minut.

```
DECLARE @ReceiveDate datetime = DATEADD(MINUTE, 15, GETDATE()),
            @OrderID int

--zamowienie dla klienta indywidualnego
EXEC dbo.AddOrder @CustomerID = 1004,
                          @EmployeeID = 20,
                          @ReceptionType = 'takeaway',
                          @PaymentType = 'in-advance',
                          @OrderDate = NULL,
                          @ReceiveDate = @ReceiveDate,
                          @Value = 0,
                          @Received = NULL,
                          @Settled = 0,
                          @OrderID = @OrderID OUTPUT
```

W rezultacie zamówienie zostało dodane do bazy.

	OrderID	CustomerID	EmployeeID	ReceptionType	PaymentType	OrderDate	ReceiveDate	Value	Discount	Completed	Received	Settled
1	1002	1004	20	takeaway	in-advance	2021-01-19 20:38:24.450	2021-01-19 20:53:24.437	0.00	NULL	0	NULL	0

Teraz dodajemy dania do zamówienia i zatwierdzamy jego zawartość, co powoduje wystawienie rachunku.

```
--dodanie dan do zamowienia
EXEC dbo.AddDishToOrder @OrderID, 100, 2
EXEC dbo.AddDishToOrder @OrderID, 300, 1
EXEC dbo.AddDishToOrder @OrderID, 400, 1
--zatwierdzenie zawartosci zamowienia
EXEC dbo.CompleteOrder 1001
```

Powstały w ten sposób rachunek wygląda następująco:

	ReceiptID	InvoicedCustomer	InvoiceDate	AccountNumber	ReceiptType	PaymentMethod	Value	Settled	Cancelled	SaleDate
1	1002	1004	2021-01-19 20:45:03.340	NULL	receipt	transfer	4.67	0	0	2021-01-19 20:45:03.340

Następnie opłacamy rachunek, co powoduje zmianę Settled na 1.

```
EXEC dbo.SettleReceipt @OrderID
```

Po wykonaniu procedury rekordy z tabel Orders oraz Receipt wyglądają następująco.

	OrderID	CustomerID	EmployeeID	ReceptionType	PaymentType	OrderDate	ReceiveDate	Value	Discount	Completed	Received	Settled
1	1002	1004	20	takeaway	in-advance	2021-01-19 20:38:24.450	2021-01-19 20:53:24.437	4.67	0.00	1	NULL	1

	ReceiptID	InvoicedCustomer	InvoiceDate	AccountNumber	ReceiptType	PaymentMethod	Value	Settled	Cancelled	SaleDate
1	1002	1004	2021-01-19 20:45:03.340	NULL	receipt	transfer	4.67	1	0	2021-01-19 20:45:03.340

Następnie zatwierdzamy odbiór zamówienia.

```
EXEC UpdateReceived @OrderID, 1
```

Powoduje to zmianę Received na 1.

	OrderID	CustomerID	EmployeeID	ReceptionType	PaymentType	OrderDate	ReceiveDate	Value	Discount	Completed	Received	Settled
1	1002	1004	20	takeaway	in-advance	2021-01-19 20:38:24.450	2021-01-19 20:53:24.437	4.67	0.00	1	1	1

W tym momencie uruchamiany jest trigger aktualizujący statystyki klienta, które później posłużą do przyznaniu klientowi rabatu.

Po wszystkich operacjach informacje o kliencie prezentują się następująco.

	IndividualCustomerID	CompanyID	FirstName	LastName	Age	PersonalDataAgreement	NumberOfOrders	OrdersInRow	SumOfOrders
1	1004	NULL	Marcin	Najman	41	1	2	0	89.94

Jak widać, wartość zamówienia została dodana do SumOfOrders, jednak nie była ona wystarczająca, aby dodać zamówienie do OrdersInRow, a także do NumberOfOrders i w konsekwencji OrdersInRow zostało wyzerowane.