Spis treści

Wstęp **6**
Ogólne informacje dotyczące systemu 6
Główne założenia **6**
Podział użytkowników systemu 6
Administrator 6
Właściciel 6
Manager 6
Pracownik 7
Niezarejestrowany Klient 7
Zarejestrowany Klient 7
Klient Firmowy 7
Główne funkcje systemu 8
Szczegóły techniczne: 8
Słowniczek: 8
Schemat **9**
Opisy Tabel **10**
ReservationVariables 10
Reservations 11
ReservationDetails 13
Tables 14
CompanyCustomerList 15
Customers 16
IndividualCustomers 17
LoyaltyCards 18
Discounts 19
OneTimeDiscounts 20
BigDiscounts 21
IndividualCustomersAddresses 22
CompanyCustomers 23
Addresses 24
Cities 25
Employees 26
EmployeesDetails 27
Orders 28
TakeawayOrders 29
OrderDetails 30
Menu 31
Meals 31
Categories 33
Informacje o danych **33**

Widoki **33**
Widok OrdersInfo 34
Widok PendingReservations 35
Widok WeeklyReservations 35
Widok MonthlyReservations 35
Widok ReservedTablesWeekly 35
Widok ReservedTablesMonthly 36
Widok MenuView 36
Widok MealsSold 37
Widok MealsSoldWeekly 37
Widok MealsSoldMonthly 37
Widok DiscountInfo 38
Widok DiscountWeekly 38
Widok DiscountMonthly 38
Widok OneTimeDiscountWeekly 39
Widok OneTimeDiscountMonthly 39
Widok OrderStatsWeekly 39
Widok OrderStatsMonthly 40
Widok OrdersAwaitingPayment 40
Widok AwaitingTakeawayOrders 40
Widok ClientsAwaitingPayments 41
Widok CustomersStatistics 41
Widok CategoriesStatistics 42
Widok LoyaltyCardsActivatedWeekly 42
Widok LoyaltyCardsActivatedMonthly 42
Widok EmptyTables 42

Typy **43**
Typ FullOrder 44
Typ GuestList 44
Typ TableList 44

Procedury **45**
Procedura AddCustomer 45
Procedura AddIndividualCustomer 45
Procedura AddCompanyCustomer 46
Procedura AddEmployee 47
Procedura ModifyEmployee 48
Procedura AddEmployeeToOrder 50
Procedura AddOrder 51
Procedura PayForOrder 54
Procedura AddOrderDetail 55
Procedura ModifyOrderDetail 56
Procedura AddMeal 56
Procedura AddMealToMenu 57

Procedura RemoveMealFromMenu 57
Procedura AddCategory 58
Procedura AddCompanyReservation 59
Procedura AddIndividualReservation 60
Procedura ChangeReservationStatus 63
Procedura AddTableToReservation 64
Procedura AddTakeawayOrder 65
Procedura AddTable 65
Procedura ModifyTable 66
Procedura ChangeAddress 67
Procedura AddCity 68
Procedura AddAddress 69
Procedura AddDiscount 69
Procedura AddBigDiscount 70
Procedura ActivateDiscountOnOrder 70
Procedura CheckCustomerDiscounts 71
Procedura RefreshMenu 73
Procedura AcceptReservation 76
Procedura RejectReservation 76
Procedura SetMenuMealProperty 76

Funkcje **78**
Funkcja GetEmployees 78
Funkcja GetFiredEmployees 78
Funkcja GetHiredEmployees 79
Funkcja GetIndividualCustomers 79
Funkcja GetCompanyCoustomers 80
Funkcja GetCoustomerById 80
Funkcja GetOrderInformations 81
Funkcja GetValueOfOrdersInDay 83
Funkcja GetValueOfOrdersInMonth 83
Funkcja GetXBestMeals 84
Funkcja GetMenuByDate 84
Funkcja GetMealsSoldAtLeastXTimes 84
Funkcja IsTodaysMenuCorrect 85
Funkcja IsMenuCorrectByDate 85
Funkcja GetClientsOrderedMoreThanXTimes 86
Funkcja GetEmployeeAddress 86
Funkcja isSeafood 86

Triggery **88**
Trigger DeleteOrderOnReservationRejection 88
Trigger ValidatePremiumMealOrdered 89

Indeksy **91**
Addresses 91

Address 91
AddressToCity 91
AddressStreet 91
AddressNumber 91
BigDiscounts 91
BigDiscount 91
Categories 92
Category 92
CategoryName 92
Cities 92
City 92
PostalCode 92
Name 92
CompanyCustomers 92
CompanyCustomer 92
CompanyNIP 93
CompanyName 93
Address 93
Customers 93
Customer 93
CustomersEmail 93
CustomersPhoneNumber 93
IndividualCustomers 93
Customer 93
FirstName 94
LastName 94
IndividualCustomersAddresses 94
CustomerAddress 94
Address 94
Discounts 94
Discount 94
Employees 94
Employee 94
EmployeesName 95
EmployeesNameReversed 95
EmployeeDetails 95
AssignedEmployee 95
LoyaltyCards 95
LoyaltyCard 95
Customer 95
Discount 95
Meals 95
Meal 96
MealName 96

Menu 96
MealInMenu 96
Meal 96
OneTimeDiscounts 96
OneTimeDiscount 96
Customer 96
BigDiscount 97
OrderDetails 97
OrderDetail 97
OrderDetailsToMenu 97
Orders 97
Order 97
OrderToReservation 97
OrderToCustomer 97
ReservationDetails 97
AssignedTable 97
Reservations 98
ReservationsCustomers 98
ReservationsCustomers 98
ReservationVariables 98
Variables 98
TakeawayOrders 98
TakeawayOrder 98
TakeawayOrderToOrder 98
Tables 98
Table 99
TablesSize 99

Uprawnienia **100**
Administrator 100
Moderator 100
Manager 100
Pracownik 100

Wstęp

Ogólne informacje dotyczące systemu

System ma na celu wspomagać firmę gastronomiczną posiadającą jedną restauracje które realizuje zamówienia na miejscu, na wynos lub internetowo. System także implementuje system rabatów dla klientów. Wspierane także będzie zmienianie menu i monitorowanie go. Ważnymi funkcjonalnościami także będą funkcje statystyczne umożliwiające analizowanie biznesu.

Główne założenia

Podział użytkowników systemu

Administrator

Osoba zarządzająca systemem. Brak ograniczeń.

- zarządzanie wszystkimi użytkownikami systemu (dodawanie / usuwanie / helpdesk)
- modyfikacja tabel (zamówienia / rezerwacje / …)
- dodawanie nowych funkcji systemu

Właściciel

Osoba mająca pełną władzę nad restauracją.

- zarządzanie pracownikami
- zarządzanie managerami
- generowanie raportów
- zarządzanie zamówieniami
- zarządzanie rezerwacjami
- zarządzanie rabatami na karcie stałego klienta
- zarządzanie menu
- zarządzanie listą potraw
- przyznawanie jednorazowych rabatów
- wystawianie faktur / paragonów
- zarządzanie cenami potraw

Manager

Osoba zatrudniona w restauracji na wyższym stanowisku.
Funkcjonalności:

- zarządzanie rezerwacjami
- zarządzanie stolikami
- zarządzanie zamówieniami
- przyznawanie jednorazowych rabatów
- wystawianie faktur / paragonów
- zarządzać pracownikami
- zarządzanie listą potraw

Pracownik

Osoba zatrudniona w restauracji.

Funkcjonalności:
- zarządzanie zamówieniami
- przyznawanie jednorazowych rabatów
- wystawianie faktur / paragonów
- aktualizowanie liczby dostępnych potraw

Niezarejestrowany Klient

Osoba kupująca w restauracji bez konta.

Funkcjonalności:
- składanie zamówienia w restauracji (na miejscu / na wynos)

Zarejestrowany Klient

Osoba kupująca w restauracji z kontem.

Funkcjonalności:
- składanie zamówienia w restauracji na wynos (odbiór w restautracji)
- składanie zamówienia online z przedpłatą (na miejscu / na wynos)
- rezerwacja stolika dla zaufanego klienta (ktoś tworzy rezerwacje + zamówienie -> akceptacja od managera -> płatność na miejscu / online (potwierdzenie minimum godzinę przed))
- karta stałego klienta do naliczania rabatów
- otrzymywanie czasowych rabatów jednorazowych
- wystawianie opinii restauracji

Klient Firmowy

Nie jest objęta systemem rabatów.

Funkcjonalności:
- rezerwacja stolika / stolików dla pracownika firmy
- rezerwacje stolika / stolików dla firmy
- różne sposoby rozliczania rezerwacji
- historia rezerwacji
- historia zamówień
- składanie zamówień online (catering)
- możliwość prowadzenia rachunku rozliczanego miesięcznie (faktura)

Główne funkcje systemu

- Wyliczenie wartości zamówienia (odpowiednio uwzględniając rabat)
- Przyznawanie rabatów jednorazowych (raz co spełnienie jakichś warunków)
- Prowadzenie karty stałego klienta
- Sprawdzenie dostępności stolików
- Sprawdzenie warunków rezerwacji stolika przez klienta
- Przypisanie klientowi numeru stolika
- Generowanie raportów

Szczegóły techniczne:

Implementacja niezarejestrowanego klienta:
W naszej bazie traktujemy niezarejestrowanego klienta jako klienta o id 0.

Słowniczek:

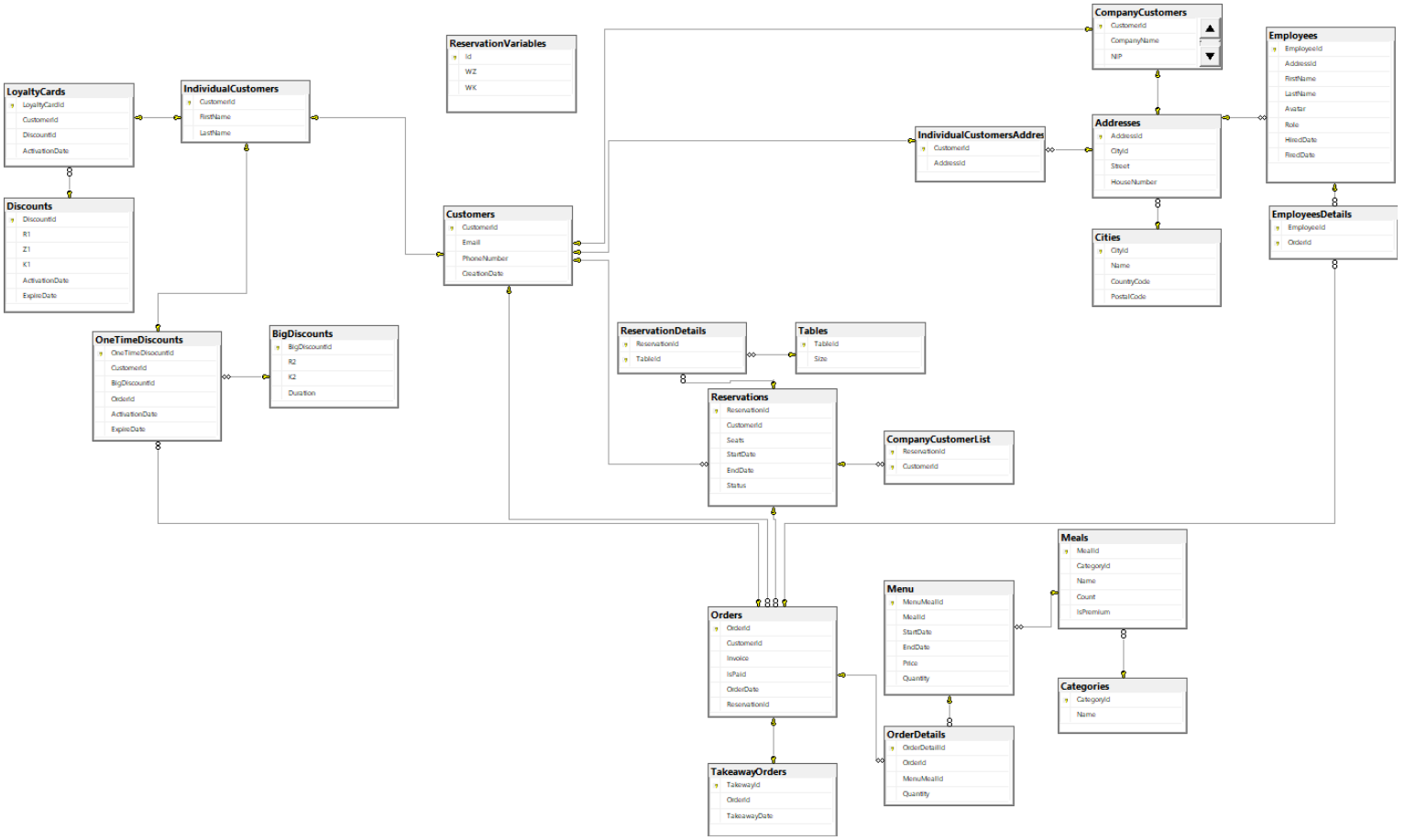
Karta stałego klienta - służy do przyznawania rabatów po spełnieniu określonych warunków, karta jest przyznawana na określony czas np. na rok, po skończeniu się przyznawana jest nowa karta która, działa na nowych warunkach.

Danie specjalne - (danie zawierające owoce morza), zamówienie dania specjalnego musi zostać złożone do poniedziałku w tygodniu w którym jest zamówienie (sam poniedziałek wlicza się do poprzedniego tygodnia)

Zarządzanie - odnosi się do dostępu operacji CRUD na odpowiedniej tabeli

Danie premium - za danie premium uznajemy m.in. danie zawierające owoce morza. Takie posiłki, zgodnie z wymaganiami, muszą zostać zamówione z odpowiednim wyprzedzeniem

Schemat


CompanyCustomers
CustomerId
CompanyName
NIP
Employees
EmployeeId
AddressId
FirstName
LastName
Avatar
Role
HiredDate
FiredDate
ReservationVariables
Id
WZ
WK
LoyaltyCards
LoyaltyCardId
CustomerId
DiscountId
ActivationDate
IndividualCustomers
CustomerId
FirstName
LastName
IndividualCustomersAddres
CustomerId
AddressId
Addresses
AddressId
CityId
Street
HouseNumber
EmployeesDetails
EmployeeId
OrderId
Discounts
DiscountId
R1
Z1
K1
ActivationDate
ExpireDate
Customers
CustomerId
Email
PhoneNumber
CreationDate
Cities
CityId
Name
CountryCode
PostalCode
OneTimeDiscounts
OneTimeDiscountId
CustomerId
BigDiscountId
OrderId
ActivationDate
ExpireDate
BigDiscounts
BigDiscountId
R2
K2
Duration
ReservationDetails
ReservationId
TableId
Tables
TableId
Size
Reservations
ReservationId
CustomerId
Seats
StartDate
EndDate
Status
CompanyCustomerList
ReservationId
CustomerId
Meals
MealId
CategoryId
Name
Count
IsPremium
Menu
MenuMealId
MealId
StartDate
EndDate
Price
Quantity
Orders
OrderId
CustomerId
Invoice
IsPaid
OrderDate
ReservationId
Categories
CategoryId
Name
OrderDetails
OrderDetailId
OrderId
MenuMealId
Quantity
TakeawayOrders
TakeawayId
OrderId
TakeawayDate

Opisy Tabel

ReservationVariables

Tabela przechowująca stałe WZ i WK. Posiada tylko pojedynczy wiersz

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	**Id**	Klucz główny tabeli ze stałymi WZ i WK.
	WZ	Minimalna wartość zamówienia.
	WK	Minimalna liczba dokonanych zamówień przez pojedynczego klienta.

Implementacja:

```
CREATE TABLE [ReservationVariables] (
 [Id] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [WZ] float NOT NULL,
 [WK] float NOT NULL
)
```

Reservations

Tabela przechowująca dokonane rezerwacje

Zmienne:

Typ	**Nazwa zmiennej**	za co odpowiada
PK	**ReservationId**	ID rezerwacji - klucz podstawowy
FK	CustomerId	ID klienta składającego rezerwację
	Seats	Liczba zarezerwowanych miejsc
	StartDate	Data i godzina początku rezerwacji
	EndDate	Data i godzina końca rezerwacji
	Status	Informacja o aktualnym statusie rezerwacji: - “**Awaiting**” - oczekuje na akceptację - “**Confirmed**” - rezerwacja potwierdzona - “**Rejected**” - rezerwacja odrzucona

Warunki Integralności:

Nazwa	Opis
R_ValidDate	Sprawdzenie, czy data końca rezerwacji następuje po dacie początku rezerwacji.
R_ValidSeats	Sprawdzenie, zarezerwowano stolik dla co najmniej 2 osób.
R_ValidStatus	Sprawdzenie, czy podano prawidłową wartość statusu.

Implementacja:

```
CREATE TABLE [Reservations] (
 [ReservationId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [CustomerId] int,
 [Seats] int NOT NULL,
 [StartDate] datetime NOT NULL,
 [EndDate] datetime NOT NULL,
 [Status] nvarchar(255) NOT NULL CHECK ([Status] IN ('Awaiting', 'Confirmed',
'Rejected')),
 CONSTRAINT R_ValidDate CHECK ( StartDate < EndDate ),
 CONSTRAINT R_ValidSeats CHECK (Seats >= 2 )
)

ALTER TABLE [Reservations] ADD FOREIGN KEY ([OrderId]) REFERENCES [Orders]
([OrderId])

ALTER TABLE [Reservations] ADD FOREIGN KEY ([CustomerId]) REFERENCES
[Customers] ([CustomerId])
```

ReservationDetails

Tabela przechowująca przypisania stołów do konkretnej rezerwacji w postaci unikalnych krotek.

Zmienne:

Typ	**Nazwa zmiennej**	za co odpowiada
PK, FK	**ReservationId**	ID rezerwacji.
PK, FK	**TableId**	ID stolika przypisanego do rezerwacji.

Warunki Integralności:

Nazwa	Opis
PK_ReservationDetails	Klucz złożony (**ReservationDetails**.ReservationId, **ReservationDetails**.TableId).

Implementacja:

```
CREATE TABLE [ReservationDetails] (
 [ReservationId] int NOT NULL,
 [TableId] int NOT NULL,
 CONSTRAINT PK_ReservationDetails PRIMARY KEY (ReservationId, TableId)
)

ALTER TABLE [ReservationDetails] ADD FOREIGN KEY ([ReservationId]) REFERENCES
[Reservations] ([ReservationId])

ALTER TABLE [ReservationDetails] ADD FOREIGN KEY ([TableId]) REFERENCES
[Tables] ([TableId])
```

Tables

Tabela przechowuje informacje o stolikach dostępnych w restauracji.

Zmienne:

Typ	**Nazwa zmiennej**	za co odpowiada
PK	TableId	ID stołu.
	Size	Maksymalna ilość miejsc przy stole.

Warunki Integralności:

Nazwa	Opis
T_ValidSize	Sprawdza, czy ilość miejsc przy stoliku jest większa od 0.

Implementacja:

```
CREATE TABLE [Tables] (
 [TableId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [Size] int NOT NULL,
 CONSTRAINT T_ValidSize CHECK (Size > 0)
)
```

CompanyCustomerList

Tabela przechowuje listę pracowników firmy, która złożyła imienną rezerwację na swoich pracowników.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK, FK	ReservationId	ID rezerwacji firmowej.
PK	CustomerId	ID klienta objętego rezerwacją firmową.

Implementacja:

```
CREATE TABLE CompanyCustomerList(
  [ReservationId] [int] NOT NULL,
  [CustomerId] [int] NOT NULL,
CONSTRAINT [PK_CompanyCustomerList] PRIMARY KEY CLUSTERED
(
  [ReservationId] ASC,
  [CustomerId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE CompanyCustomerList  WITH CHECK ADD  CONSTRAINT
[FK_CompanyCustomerList_Reservations] FOREIGN KEY([ReservationId])
REFERENCES Reservations ([ReservationId])
GO

ALTER TABLE CompanyCustomerList CHECK CONSTRAINT
[FK_CompanyCustomerList_Reservations]
GO
```

Customers

Tabela zawierająca informacje ogólne zarówno o klientach indywidualnych, jak i firmowych.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	**CustomerId**	ID klienta.
	Email	Adres e-mail klienta.
	PhoneNumber	Numer telefonu klienta.
	CreationDate	Data utworzenia konta klienta.

Warunki Integralności:

nazwa	Opis
CC_ValidEmail	Sprawdza poprawność wprowadzonego adresu e-mail.
CC_ValidPhoneNumber	Sprawdza poprawność wprowadzonego numeru telefonu.

Implementacja:

```
CREATE TABLE [Customers] (
 [CustomerId] int PRIMARY KEY NOT NULL IDENTITY(100, 1),
 [Email] varchar(255) NOT NULL UNIQUE,
 [PhoneNumber] varchar(15) NOT NULL UNIQUE,
 [CreationDate] datetime NOT NULL,
 CONSTRAINT CC_ValidEmail CHECK (Email LIKE '%@%'),
 CONSTRAINT CC_ValidPhoneNumber CHECK (SUBSTRING (PhoneNumber, 1, 1) LIKE
'[0-9+]' AND ISNUMERIC(PhoneNumber) = 1)
)
```

IndividualCustomers

Tabela zawierająca informacje dotyczące klienta indywidualnego.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada
PK, FK	**CustomerId**	ID klienta.
	FirstName	Imię klienta.
	LastName	Nazwisko klienta.

Implementacja:

```
CREATE TABLE [IndividualCustomers] (
 [CustomerId] int PRIMARY KEY NOT NULL,
 [FirstName] varchar(64) NOT NULL,
 [LastName] varchar(128) NOT NULL
)

ALTER TABLE [IndividualCustomers] ADD FOREIGN KEY ([CustomerId]) REFERENCES
[Customers] ([CustomerId])
```

LoyaltyCards

Tabela przechowuje aktualnie aktywną kartę lojalnościową klienta. Każdy klient w momencie utworzenia konta ma zakładaną kartę lojalnościową z polem ActivationDate ustawionym na NULL.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	LoyaltyCardId	Id karty klienta.
FK	CustomerId	Id klienta, do którego karta należy.
FK	DiscountId	Id rabatu na karcie - daje informację, który rabat obowiązuje danego klienta.
	ActivationDate	Data od której naliczane są rabaty do zamówień. Data ustawiana jest po spełnieniu warunków Z1 i K1.

Implementacja:

```
CREATE TABLE [LoyaltyCards] (
 [LoyaltyCardId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [CustomerId] int NOT NULL,
 [DiscountId] int NOT NULL,
 [ActivationDate] datetime,
)

ALTER TABLE [LoyaltyCards] ADD FOREIGN KEY ([CustomerId]) REFERENCES
[IndividualCustomers] ([CustomerId])

ALTER TABLE [LoyaltyCards] ADD FOREIGN KEY ([DiscountId]) REFERENCES
[Discounts] ([DiscountId])
```

Discounts

Tabela przechowuje dane rabatów załączanych do karty lojalnościowej.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	**DiscountId**	ID rabatu - klucz podstawowy.
	R1	Otrzymywany rabat po spełnieniu warunków Z1 oraz K1.
	Z1	Minimalna liczba zamówień, jaką trzeba złożyć, aby otrzymać rabat w wysokości R1.
	K1	Minimalna wartość zamówienia zaliczającego się do liczenia zamówień Z1.
	ActivationDate	Czas utworzenia rabatu.
	ExpireDate	Czas wyłączenia rabatu.

Implementacja:

```
CREATE TABLE [Discounts] (
 [DiscountId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [R1] float NOT NULL,
 [Z1] int NOT NULL,
 [K1] money NOT NULL,
 [ActivationDate] datetime NOT NULL,
 [ExpireDate] datetime
)
```

OneTimeDiscounts

Tabela przechowuje informacje o zniżce jednorazowej.

Zmienne:

Typ	**Nazwa zmiennej**	za co odpowiada
PK	OneTimeDiscountId	ID rabatu jednorazowego R2.
FK	CustomerId	ID klienta, którego dotyczy zniżka.
FK	BigDiscountId	ID zniżki, obowiązującej klienta.
FK	OrderId	ID zamówienia, przy którym została użyta zniżka, jeśli nie została użyta, to pole ma wartość NULL.
	ActivationDate	Data rozpoczęcia obowiązywania zniżki.
	ExpireDate	Data zakończenia obowiązywania zniżki.

Implementacja:

```
CREATE TABLE [OneTimeDiscounts] (
 [OneTimeDisocuntId] int PRIMARY KEY NOT NULL IDENTITY(1,1),
 [CustomerId] int NOT NULL,
 [BigDiscountId] int NOT NULL,
 [OrderId] int,
 [ActivationDate] datetime,
 [ExpireDate] datetime,
)

ALTER TABLE [OneTimeDiscounts] ADD FOREIGN KEY ([BigDiscountId])
REFERENCES [BigDiscounts] ([BigDiscountId])

ALTER TABLE [OneTimeDiscounts] ADD FOREIGN KEY ([CustomerId])
REFERENCES [IndividualCustomers] ([CustomerId])

ALTER TABLE [OneTimeDiscounts] ADD FOREIGN KEY ([OrderId])
REFERENCES [Orders] ([OrderId])
```

BigDiscounts

Tabela przechowuje dane o rabatach jednorazowych (R2). Tabela pełni rolę słownika.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	**Id**	ID rabatu.
	R2	Otrzymywany rabat po spełnieniu warunku K2.
	K2	Łączna kwota wydana na zamówienia, od której aktywuje się rabat R2.
	Duration	Czas trwania rabatu R2 (w dniach).

Implementacja:

```
CREATE TABLE [BigDiscounts] (
 [BigDiscountId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [R2] float NOT NULL,
 [K2] money NOT NULL,
 [Duration] INT NOT NULL,
)
```

IndividualCustomersAddresses

Tabela zawierająca krotki w postaci ID klienta, ID jego adresu. Dane te są potrzebne do faktury.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK, FK	**CustomerId**	ID klienta.
FK	AddressId	ID adresu klienta.

Implementacja:

```
CREATE TABLE [IndividualCustomersAddresses] (
   [CustomerId] INT PRIMARY KEY NOT NULL,
   [AddressId] int NOT NULL,
)

ALTER TABLE [IndividualCustomersAddresses] ADD FOREIGN KEY ([CustomerId])
REFERENCES [Customers] ([CustomerId])

ALTER TABLE [IndividualCustomersAddresses] ADD FOREIGN KEY ([AddressId])
REFERENCES [Addresses] ([AddressId])
```

CompanyCustomers

Tabela zawierająca informacje dotyczące klienta firmowego.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK, FK	**CustomerId**	ID klienta firmowego.
	CompanyName	Nazwa firmy.
	NIP	NIP firmy.
FK	AddressId	ID adresu firmy.

Warunki Integralności:

nazwa	Opis
CC_ValidNIP	Sprawdzenie poprawności numeru NIP.

Implementacja:

```
CREATE TABLE [CompanyCustomers] (
 [CustomerId] int PRIMARY KEY NOT NULL,
 [CompanyName] varchar(255) NOT NULL,
 [NIP] varchar(32) NOT NULL,
 [AddressId] int NOT NULL,
 CONSTRAINT CC_ValidNIP CHECK (NIP LIKE
'[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]')
)

ALTER TABLE [CompanyCustomers] ADD FOREIGN KEY ([CustomerId]) REFERENCES
[Customers] ([CustomerId])

ALTER TABLE [CompanyCustomers] ADD FOREIGN KEY ([AddressId]) REFERENCES
[Addresses] ([AddressId])
```

Addresses

Tabela przechowuje dane adresowe klientów. Każdy klient, który podał adres oraz każdy klient firmowy ma swój własny AddressId.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	AddressId	ID adresu.
FK	CityId	ID miasta.
	Street	Nazwa ulicy przy której klient mieszka / przy której firma ma siedzibę.
	HouseNumber	Numer domu / mieszkania / budynku.

Implementacja:

```
CREATE TABLE [Addresses] (
 [AddressId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [CityId] int NOT NULL,
 [Street] varchar(255) NOT NULL,
 [HouseNumber] varchar(32) NOT NULL
)

ALTER TABLE [Addresses] ADD FOREIGN KEY ([CityId]) REFERENCES [Cities]
([CityId])
```

Cities

Tabela przechowuje informacje o mieście zamieszkania pracowników oraz klientów, a w przypadku klienta firmowego - o mieście, w którym firma ma główną siedzibę.

Zmienne:

Typ	**Nazwa zmiennej**	za co odpowiada
PK	CityId	ID miasta.
	Name	Nazwa miasta.
	CountryCode	Kod kraju, w którym dane miasto się znajduje.
	PostalCode	Kod pocztowy danej miejscowości.

Implementacja:

```
CREATE TABLE [Cities] (
 [CityId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [Name] varchar(255) NOT NULL,
 [CountryCode] varchar(2) NOT NULL
 [PostalCode] varchar(5) NOT NULL
)
```

Employees

Tabela przechowuje informacje o pracownikach restauracji.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	EmployeeId	ID pracownika.
FK	AddressId	ID adresu zameldowania.
	FirstName	Imię pracownika.
	LastName	Nazwisko pracownika.
	Avatar	URL do zdjęcia pracownika.
	Role	Rola pracownika - Worker / Manager / Owner.
	HiredDate	Data zatrudnienia pracownika.
	FiredDate	Data zwolnienia pracownika.

Warunki Integralności:

Nazwa	Opis
CK_Role	Ogranicza zbiór wartości pola Role do ‘Worker’, ‘Manager’, ‘Owner’.

Implementacja:

```
CREATE TABLE [Employees] (
 [EmployeeId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [AddressId] int NOT NULL,
 [FirstName] varchar(32) NOT NULL,
 [LastName] varchar(64) NOT NULL,
 [Avatar] varchar(255),
 [Role] nvarchar(255) NOT NULL CHECK ([Role] IN ('Worker',
'Manager', 'Owner')),
 [HiredDate] datetime NOT NULL,
 [FiredDate] datetime
)

ALTER TABLE [Employees] ADD FOREIGN KEY ([AddressId]) REFERENCES [Addresses]
([AddressId])
```

EmployeesDetails

Tabela przechowywująca przypisania pracowników do obsługi konkretnego zamówienia w postaci unikalnych krotek.

Zmienne:

Typ	**Nazwa zmiennej**	za co odpowiada
PK, FK	**EmployeeId**	ID pracownika obsługującego zamówienie.
PK, FK	**OrderId**	ID zamówienia, do którego został przypisany pracownik o danym EmployeeId.

Warunki Integralności:

nazwa	opis
PK_EmployeesDetails	Klucz złożony (**Employees**.EmployeeId, **Orders**.OrderId)

Implementacja:

```
CREATE TABLE [EmployeesDetails] (
 [EmployeeId] int NOT NULL,
 [OrderId] int NOT NULL,
 CONSTRAINT PK_EmployeesDetails PRIMARY KEY (Employeeid, OrderId)
)

ALTER TABLE [EmployeesDetails] ADD FOREIGN KEY ([OrderId]) REFERENCES
[Orders] ([OrderId])

ALTER TABLE [EmployeesDetails] ADD FOREIGN KEY ([EmployeeId]) REFERENCES
[Employees] ([EmployeeId])
```

Orders

Tabela przechowuje informacje o zamówieniach.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	OrderId	ID zamówienia.
FK	CustomerId	ID klienta, który złożył zamówienie. Dotyczy zarówno klienta indywidualnego, jak i klienta firmowego.
	Invoice	Informacja, czy klient zażyczył sobie faktury do zamówienia.
	IsPaid	Informacja, czy zamówienie zostało już opłacone.
	OrderDate	Data złożenia zamówienia.
FK	ReservationId	ID rezerwacji, jeżeli została wcześniej złożona, w przeciwnym wypadku pole przyjmuje wartość NULL.

Warunki Integralności:

Nazwa	Opis
DF_Invoice	Ustawia domyślną wartość przy polu “Faktura” na false (0).
DF_IsPaid	Ustawia domyślną wartość przy polu IsPaid na false (0).
DF_OrderDate	Ustawia domyślną datę zamówienia na teraźniejszą.

Implementacja:

```
CREATE TABLE [Orders] (
 [OrderId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [CustomerId] int NOT NULL,
 [Invoice] bit default 0 NOT NULL,
 [IsPaid] bit default 0 NOT NULL,
 [OrderDate] datetime NOT NULL DEFAULT GETDATE(),
 [ReservationId] int,
)

ALTER TABLE [Orders] ADD FOREIGN KEY ([CustomerId]) REFERENCES
[Customers] ([CustomerId])

ALTER TABLE [Orders] ADD FOREIGN KEY ([ReservationsId]) REFERENCES
[Reservations] ([ReservationsId])
```

TakeawayOrders

Tabela przechowuje informacje o zamówieniach na wynos.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	TakeawayId	ID zamówienia na wynos - klucz podstawowy.
FK	OrderId	ID zamówienia.
	TakeawayDate	Data odbioru zamówienia na wynos.

Warunki Integralności:

Nazwa	Opis
TO_ValidDate	Sprawdza, czy data odbioru zamówienia jest późniejsza niż aktualna data.

Implementacja:

```
CREATE TABLE [TakeawayOrders] (
 [TakewayId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [OrderId] int NOT NULL,
 [TakeawayDate] datetime NOT NULL,
 CONSTRAINT TO_ValidDate CHECK (TakeawayDate > GETDATE())
)

ALTER TABLE [TakeawayOrders] ADD FOREIGN KEY ([OrderId])
REFERENCES [Orders] ([OrderId])
```

OrderDetails

Tabela przechowuje szczegóły zamówień - co zostało zamówione i w jakiej ilości.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	**OrderDetailId**	ID historii zamówienia - klucz podstawowy.
FK	OrderId	ID zamówienia.
FK	MenuMealId	ID dania znajdującego się w menu.
	Quantity	Liczba zamówionych jednostek dania przez klienta.

Warunki Integralności:

Nazwa	Opis
OD_ValidQuantity	Sprawdza czy liczpa podanych jendostek zamówienia jest poprawna (wieksza od 0).
DF_Quantity	Ustawia domyślną ilość zamówionych jednostek posiłku na 1.

Implementacja:

```
CREATE TABLE [OrderDetails] (
 [OrderDetailId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [OrderId] int NOT NULL,
 [MenuMealId] int NOT NULL,
 [Quantity] int NOT NULL default 1,
  CONSTRAINT OD_ValidQuantity Check (Quantity > 0)
)

ALTER TABLE [OrderDetails] ADD FOREIGN KEY ([OrderId]) REFERENCES [Orders]
([OrderId])

ALTER TABLE [OrderDetails] ADD FOREIGN KEY ([MenuMealId]) REFERENCES
[Menu] ([MenuMealId])
```

Menu

Tabela przechowuje informacje o wszystkich posiłkach, które kiedykolwiek znalazły się w menu.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	MenuMealId	ID posiłku w menu - klucz podstawowy.
FK	MealId	ID posiłku.
	StartDate	Informacja o tym, kiedy dany posiłek został dodany do obowiązującego wówczas menu.
	EndDate	Informacja, o tym do kiedy posiłek był obecny w obowiązującym menu, wartość NULL oznacza, że posiłek znajduje się obecnie w menu.
	Price	Cena za posiłek, która obowiązywała w okresie [StartDate, EndDate].
	Quantity	Całkowita liczba dostępnych posiłków (wartość nie może być zmniejszona)

Warunki Integralności:

Nazwa	Opis
MN_ValidPrice	Sprawdza, czy cena jest większa lub równa 0 zł.
MMN_ValidQuantity	Sprawdzenie, czy liczba dostępnych dań jest nieujemna.

Implementacja:

```
CREATE TABLE [Menu] (
 [MenuMealId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [MealId] int NOT NULL,
 [StartDate] datetime NOT NULL,
 [EndDate] datetime,
 [Price] money NOT NULL,
 [Quantity] int,
 CONSTRAINT MN_ValidPrice CHECK ( Price >= 0 ),
 CONSTRAINT MN_ValidQuantity CHECK (Quantity >= 0)
)

ALTER TABLE [Menu] ADD FOREIGN KEY ([MealId]) REFERENCES [Meals]
([MealId])
```

Meals

Tabela, będąca słownikiem na wszystkie dostępne dania w restauracji.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	MealId	ID posiłku - klucz podstawowy.
FK	CategoryId	ID kategorii.
	Name	Nazwa dania.
	Count	Liczba dostępnych porcji.
	IsPremium	Czy dane danie jest produktem premium.

Implementacja:

```
CREATE TABLE [Meals] (
 [MealId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [CategoryId] int NOT NULL,
 [Name] varchar(255) NOT NULL,
 [Count] int NOT NULL DEFAULT 0,
 [IsPremium] bit NOT NULL,
)

ALTER TABLE [Meals] ADD FOREIGN KEY ([CategoryId]) REFERENCES
[Categories] ([CategoryId])
```

Categories

Tabela przechowuje informacje o kategoriach posiłków.

Zmienne:

Typ	**Nazwa zmiennej**	Za co odpowiada?
PK	CategoryId	ID kategorii.
	Name	Nazwa kategorii.

Implementacja:

```
CREATE TABLE [Categories] (
 [CategoryId] int PRIMARY KEY NOT NULL IDENTITY(1, 1),
 [Name] varchar(255) NOT NULL,
)
```

Informacje o danych

Dane do tabel zostały wygenerowane z wykorzystaniem różnych generatorów danych, w tym z ChatGPT. Każda tabela posiada odpowiednią ilość danych, niezbędną do sprawdzenia prawidłowego działania systemu. Wszystkie dane zostały przetestowane pod kątem poprawności i zgodności z wymaganiami projektu.

Widoki

Widok OrdersInfo

Wyświetla informacje o wszystkich zamówieniach takie jak kwota czy data złożenia

```
CREATE VIEW OrdersInfo AS
(
SELECT O.OrderId                                AS OrderId,
      O.CustomerId                             as CustomerId,
      O.OrderDate                              AS OrderDate,
      ISNULL(SUM(OD.Quantity * M.Price), 0) AS Value
FROM Orders AS O
        JOIN OrderDetails AS OD on O.OrderId = OD.OrderId
        JOIN Menu M on M.MenuMealId = OD.MenuMealId
        JOIN LoyaltyCards LC on LC.CustomerId = O.CustomerId
WHERE (LC.ActivationDate IS NULL OR LC.ActivationDate > O.OrderDate)
 AND NOT EXISTS(SELECT * FROM OneTimeDiscounts AS OTD where OTD.OrderId =
O.OrderId)
GROUP BY O.OrderId, O.CustomerId, O.OrderDate
UNION
SELECT O.OrderId                                          as OrderId,
      O.CustomerId                                       as CustomerId,
      O.OrderDate                                        as OrderDate,
      ISNULL(SUM(OD.Quantity * M.Price * (1 - D.R1)), 0) as Value
FROM Orders AS O
        JOIN OrderDetails AS OD on O.OrderId = OD.OrderId
        JOIN Menu M on M.MenuMealId = OD.MenuMealId
        JOIN LoyaltyCards LC on LC.CustomerId = O.CustomerId
        JOIN Discounts D on D.DiscountId = LC.DiscountId
WHERE LC.ActivationDate IS NOT NULL
 AND LC.ActivationDate < O.OrderDate
 AND NOT EXISTS(SELECT * FROM OneTimeDiscounts AS OTD where OTD.OrderId =
O.OrderId)
GROUP BY O.OrderId, O.CustomerId, O.OrderDate
UNION
SELECT O.OrderId                                          as OrderId,
      O.CustomerId                                       as CustomerId,
      O.OrderDate                                        as OrderDate,
      ISNULL(SUM(OD.Quantity * M.Price * (1 - BD.R2)), 0) as Value
FROM Orders AS O
        JOIN OrderDetails AS OD on O.OrderId = OD.OrderId
        JOIN Menu M on M.MenuMealId = OD.MenuMealId
        JOIN OneTimeDiscounts OTD on O.OrderId = OTD.OrderId
        JOIN BigDiscounts BD on BD.BigDiscountId = OTD.BigDiscountId
GROUP BY O.OrderId, O.CustomerId, O.OrderDate)
```

Widok PendingReservations

Wyświetla rezerwacje oczekujące na akceptację.

```
CREATE VIEW PendingReservations as
select * from Reservations
where status='Awaiting'
```

Widok WeeklyReservations

Wyświetla liczbę rezerwacji z podziałem na rok i numer tygodnia w roku.

```
CREATE view [dbo].[WeeklyReservations] as
select
   datepart(year, StartDate) as year,
   datepart(week, StartDate) as week,
   count(*) as reservations
from Reservations
group by
   datepart(year, StartDate),
   datepart(week, StartDate)
with rollup
GO
```

Widok MonthlyReservations

Wyświetla liczbę rezerwacji z podziałem na rok i numer miesiąca w roku.

```
create view [dbo].[MonthlyReservations] as
select
   datepart(year, StartDate) as year,
   datepart(month, StartDate) as month,
   count(*) as reservations from Reservations
group by
   datepart(year, StartDate),
   datepart(month, StartDate)
with rollup
GO
```

Widok ReservedTablesWeekly

Wyświetla informacje ile razy dany stolik był rezerwowany z podziałem na rok i tydzień

```
CREATE VIEW ReservedTablesWeekly AS
SELECT
   T.TableId AS "Numer stolika",
   COUNT(T.TableId) AS "Liczba rezerwacji",
   DATEPART(year, R.StartDate) AS "Rok",
   DATEPART(week, R.StartDate) AS "Numer tygodnia"
```

```
FROM Tables AS T
   INNER JOIN ReservationDetails AS RD ON T.TableId = RD.TableId
   INNER JOIN Reservations AS R ON R.ReservationId = RD.ReservationId
GROUP BY T.TableId, DATEPART(year, R.StartDate), DATEPART(week, R.StartDate)
```

Widok ReservedTablesMonthly

Wyświetla informacje ile dany stolik był rezerwowany z podziałem na rok i miesiąc.

```
CREATE VIEW ReservedTablesMonthly AS
SELECT
   T.TableId AS "Numer stolika",
   COUNT(T.TableId) AS "Liczba rezerwacji",
   DATEPART(year, R.StartDate) AS "Rok",
   DATEPART(month, R.StartDate) AS "Numer miesiąca"
FROM Tables AS T
   INNER JOIN ReservationDetails AS RD ON T.TableId = RD.TableId
   INNER JOIN Reservations AS R ON R.ReservationId = RD.ReservationId
GROUP BY T.TableId, DATEPART(year, R.StartDate), DATEPART(month, R.StartDate)
```

Widok MenuView

Wyświetla aktualne menu, podstawowe informacje o potrawach oraz liczbę potraw możliwą do kupienia.

```
CREATE OR ALTER VIEW MenuView AS
SELECT M.MenuMealId,
      Meals.Name,
      M.StartDate AS [In Menu Since],
      M.Price,
      M.Quantity as [Total],
      M.Quantity - ISNULL(SUM(OD.Quantity),0) as [Avaliable]
FROM Menu as M
   JOIN Meals ON M.MealId = Meals.MealId
   LEFT JOIN OrderDetails OD on M.MenuMealId = OD.MenuMealId
WHERE M.StartDate < GETDATE()
                AND (M.EndDate IS NULL OR
                     M.EndDate > GETDATE())
GROUP BY M.MenuMealId, Meals.Name, M.StartDate, M.Price,
M.Quantity
```

Widok MealsSold

Wyświetla nazwę dania, kategorie oraz ile razy został sprzedany dla wszystkich potraw w menu.

```
CREATE VIEW MealsSold AS
SELECT 'Meal'=M.Name, 'Category'=C.Name, 'Sold units' =
isnull(SUM(OD.Quantity),0)
FROM Meals AS M
       LEFT JOIN Categories C on C.CategoryId = M.CategoryId
       LEFT JOIN Menu M2 on M.MealId = M2.MealId
       LEFT JOIN OrderDetails OD on M2.MenuMealId = OD.MenuMealId
GROUP BY M.MealId, M.Name, C.CategoryId, C.Name
```

Widok MealsSoldWeekly

Wyświetla danie oraz liczbę jego sprzedanych jednostek z podziałem na rok i tydzień.

```
CREATE VIEW MealsSoldWeekly AS
SELECT YEAR(O.OrderDate)             AS Year,
      DATEPART(week, O.OrderDate)    AS Week,
      M.Name,
      ISNULL(SUM(OD.Quantity), 0) AS [Sold Units]
FROM Meals AS M
   JOIN Menu M2 ON M.MealId = M2.MealId
   JOIN OrderDetails AS OD ON M2.MenuMealId = OD.MenuMealId
   JOIN Orders O on O.OrderID = OD.OrderID
GROUP BY YEAR(O.OrderDate), DATEPART(week, O.OrderDate), M.Nam
```

Widok MealsSoldMonthly

Wyświetla danie oraz liczbę jego sprzedanych jednostek z podziałem na rok i miesiąc.

```
CREATE VIEW MealsSoldMonthly AS
SELECT YEAR(O.OrderDate)              AS Year,
      DATEPART(month , O.OrderDate) AS Month,
      M.Name,
      ISNULL(SUM(OD.Quantity), 0) AS [Sold Units]
FROM Meals AS M
  JOIN Menu M2 on M.MealId = M2.MealId
  JOIN OrderDetails AS OD ON M2.MenuMealId = OD.MenuMealId
  JOIN Orders O on O.OrderID = OD.OrderID
GROUP BY YEAR(O.OrderDate), DATEPART(month, O.OrderDate), M.Name
```

Widok DiscountInfo

Dla każdego klienta wyświetla informacje o aktualnie przysługiwanych zniżkach powtarzających się (R1, Z1, K1) oraz czy zostały aktywowane.

```
create view DiscountInfo as
select CustomerId, active, k1, R1 from (
   select
       Customers.CustomerId,
       case when (
           isnull(
               LC.ActivationDate,
               dateadd(year, 1, getdate())
               ) < GETDATE()
           )
               then cast(1 as bit)
               else cast(0 as bit)
           end
       as active,
       D.Z1,
       D.R1,
       D.K1
   from Customers
   left join IndividualCustomers IC on Customers.CustomerId =
IC.CustomerId
   left join LoyaltyCards LC on IC.CustomerId = LC.CustomerId
   left join Discounts D on LC.DiscountId = D.DiscountId
) discountInfo
```

Widok DiscountWeekly

Wyświetla liczbę zniżek (wielorazowych) przyznanych z podziałem na rok i tydzień

```
CREATE VIEW DiscountWeekly as
select
   datepart(year, ActivationDate) as year,
   datepart(week, ActivationDate) as week,
   count(*) as discounts from LoyaltyCards
where ActivationDate is not null
group by
   datepart(year, ActivationDate),
   datepart(week, ActivationDate)
```

Widok DiscountMonthly

Wyświetla liczbę zniżek (wielorazowych) przyznanych z podziałem na rok i miesiąc.

```
CREATE VIEW DiscountMonthly as
select
```

```
   datepart(year, ActivationDate) as year,
   datepart(month, ActivationDate) as week,
   count(*) as discounts from LoyaltyCards
where ActivationDate is not null
group by
   datepart(year, ActivationDate),
   datepart(month , ActivationDate)
```

Widok OneTimeDiscountWeekly

Wyświetla liczbę zniżek jednorazowych przyznanych z podziałem na rok i tydzień

```
SELECT COUNT(*) AS "Liczba przyznanych zniżek",
   DATEPART(year, ActivationDate) AS "Rok",
   DATEPART(week, ActivationDate) AS "Numer tygodnia"
FROM OneTimeDiscounts
WHERE ActivationDate IS NOT NULL
GROUP BY DATEPART(year, ActivationDate),
   DATEPART(week, ActivationDate)
```

Widok OneTimeDiscountMonthly

Wyświetla liczbę zniżek jednorazowych przyznanych z podziałem na rok i miesiąc.

```
CREATE VIEW OneTimeDiscountMonthly AS
SELECT COUNT(*) AS "Liczba przyznanych zniżek",
    DATEPART(year, ActivationDate) AS "Rok",
    DATEPART(month, ActivationDate) AS "Numer miesiąca"
FROM OneTimeDiscounts
WHERE ActivationDate IS NOT NULL AND ExpireDate < GETDATE()
GROUP BY DATEPART(year, ActivationDate),
    DATEPART(month, ActivationDate)
```

Widok OrderStatsWeekly

Wyświetla liczbę zamówień oraz przychody z podziałem na rok i tydzień.()

```
CREATE VIEW OrderStatsWeekly AS
SELECT YEAR(OC.OrderDate) as year,
     DATEPART(week, OC.OrderDate) as week,
     SUM(OC.Value) as Income,
     COUNT(*) as OrdersNumber
FROM OrdersInfo as OC
GROUP BY YEAR(OC.OrderDate), DATEPART(week, OC.OrderDate)
```

Widok OrderStatsMonthly

Wyświetla liczbę zamówień oraz przychody z podziałem na rok i miesiąc.

```
CREATE VIEW OrderStatsMonthly AS
SELECT YEAR(OC.OrderDate) as year,
     DATEPART(month, OC.OrderDate) as month,
     SUM(OC.Value) as Income,
     COUNT(*) as OrdersNumber
FROM OrdersInfo as OC
GROUP BY YEAR(OC.OrderDate), DATEPART(month, OC.OrderDate)
```

Widok OrdersAwaitingPayment

Wyświetla liczbę zamówienia oczekujące płatności, klienta i kwotę.

```
create view OrdersAwaitingPayment as
select
   orderid,
   selected.customerid,
   round(case when (DiscountInfo.active = 1 and k1 <= price)
       then price * (1 - r1)
       else price
   end, 2) as cost
from (
   select
       orders.orderid,
       orders.CustomerId,
       isnull(sum( details.Quantity * menu.Price ), 0) as price
   from orders
    left join OrderDetails as details
       on orders.OrderId=details.OrderId
    left join Menu
       on details.MenuMealId = Menu.MenuMealId
   where orders.IsPaid=0
   group by orders.orderid, orders.CustomerId
) selected inner join DiscountInfo
on selected.CustomerId= DiscountInfo.CustomerId
```

Widok AwaitingTakeawayOrders

Wyświetla zamówienia które oczekują odbioru

```
CREATE VIEW AwaitingTakeawayOrders AS
```

```
SELECT T.TakewayId, T.OrderId, T.TakeawayDate, OI.OrderDate,
OI.Value, OI.CustomerId, OI.IsPaid
FROM TakeawayOrders as T
   JOIN OrdersInfo OI on T.OrderId = OI.OrderId
WHERE TakeawayDate > GETDATE()
```

Widok ClientsAwaitingPayments

Dla każdego klienta który nie opłacił jakiegoś zamówienia wyświetla liczbę zamówień do opłaty i ich całkowitą wartość.

```
CREATE VIEW ClientsAwaitingPayments AS
SELECT OI.CustomerId            as [Customer Id],
      ISNULL(SUM(OI.Value), 0) as MoneyToPay,
      ISNULL(COUNT(*), 0)      as OrdersToPay
FROM Customers as C
        JOIN OrdersInfo as OI ON OI.CustomerId = C.CustomerId
where OI.IsPaid = 'false'
GROUP BY OI.CustomerId
```

Widok CustomersStatistics

Wyświetla dla każdego klienta łączną kwotę (do zapłaty + opłacone), którą wydał w restauracji i liczbę zamówień.

```
CREATE VIEW CustomerStatistics AS

SELECT C.CustomerId as [Customer Id],
    SUM(OI.Value) as MoneySpent,
    COUNT(*) as OrdersNumber
FROM Customers as C
   JOIN OrdersInfo as OI ON OI.CustomerId = C.CustomerId
GROUP BY C.CustomerId
UNION
SELECT C.CustomerId as [Customer Id],
    0 as MoneySpent,
    0 as OrdersNumber
FROM Customers as C
  LEFT JOIN OrdersInfo as OI ON OI.CustomerId = C.CustomerId
GROUP BY C.CustomerId
having SUM(OI.Value) IS NULL
```

Widok CategoriesStatistics

Wyświetla kategorie i liczbę sprzedanych posiłków z danej kategorii.

```
CREATE VIEW CategoriesStatistics AS
SELECT C.Name, SUM(ISNULL(Od.Quantity,0))
FROM Categories AS C
   LEFT JOIN Meals AS M ON C.CategoryId = M.CategoryId
   LEFT JOIN Menu AS Mn ON Mn.MealId = M.MealId
   LEFT JOIN OrderDetails AS Od ON Od.MenuMealId = Mn.MenuMealId
GROUP BY C.CategoryId, C.Name
```

Widok LoyaltyCardsActivatedWeekly

Wyświetla liczba aktywowanych rabatów tygodniowo.

```
CREATE VIEW LoyaltyCardsActivatedWeek AS
SELECT YEAR(LC.ActivationDate)            AS Year,
       DATEPART(week , LC.ActivationDate) AS Week,
       ISNULL(COUNT(*), 0) AS [Activated Discounts]
FROM LoyaltyCards AS LC
WHERE LC.ActivationDate IS NOT NULL
GROUP BY YEAR(LC.ActivationDate), DATEPART(week, LC.ActivationDate)
```

Widok LoyaltyCardsActivatedMonthly

Wyświetla liczbę aktywowanych rabatów miesięcznie.

```
CREATE VIEW LoyaltyCardsActivatedMonthly AS
SELECT YEAR(LC.ActivationDate)             AS Year,
      DATEPART(month , LC.ActivationDate) AS Month,
      ISNULL(COUNT(*), 0) AS [Activated Discounts]
FROM LoyaltyCards AS LC
WHERE LC.ActivationDate IS NOT NULL
GROUP BY YEAR(LC.ActivationDate), DATEPART(month,
LC.ActivationDate)
```

Widok EmptyTables

Wyświetla id wolnych stolików (które nie są zarezerwowane fizycznie w restauracji istnieje szansa że są zajęte).

```
CREATE VIEW EmptyTables AS
SELECT TableId FROM Tables
EXCEPT
SELECT T.TableId
FROM Reservations R
  INNER JOIN ReservationDetails Rd ON R.ReservationId=
Rd.ReservationId
  INNER JOIN Tables T ON Rd.TableId = T.TableId
WHERE R.StartDate < GETDATE() AND GETDATE() < R.EndDate AND
R.Status = 'Confirmed'
```

Typy

Typ FullOrder

Typ przeznaczony dla funkcji AddOrder, służy do przekazania listy zamówień

```
create type FullOrder as table
(
   quantity int,
   mealId   int
)
```

Typ GuestList

Typ przeznaczony do przekazywania listy gości

```
create type GuestList as table
(
   CustomerId int
)
```

Typ TableList

Typ przeznaczony do przekazywania listy stolików do procedór i funkcji

```
create type TableList as table
(
   TableId int
)
```

Procedury

Procedura AddCustomer

Procedura służąca do dodania nowego klienta do tabeli `Customers`.
Uwaga! Funkcja ta nie precyzuje, czy mamy doczynenia z klientem indywidualnym, czy firmowym - w tym celu wykorszystujemy dwie następne procedury

```
CREATE OR ALTER PROCEDURE AddCustomer
  @email varchar(255),
  @phoneNumber varchar(15),
  @customerId INT = NULL OUTPUT
AS
BEGIN
  INSERT INTO Customers(Email, PhoneNumber, CreationDate) VALUES
(@email, @phoneNumber, GETDATE())

  SET @customerId = SCOPE_IDENTITY()
END
```

Procedura AddIndividualCustomer

Procedura służąca do dodania nowego klienta indywidualnego. Dla nieistniejącego klienta wewnątrz tabeli `Customers`, tworzy dla niego nowy, odpowiedni wpis

```
CREATE OR ALTER PROCEDURE AddIndividualCustomer
  @firstName varchar(64),
  @lastName varchar(128),
  @customerId INT = NULL OUTPUT ,
  @email varchar(255),
  @phoneNumber varchar(15),
  @addressId INT = NULL,
  @street varchar(255) = NULL,
  @houseNumber varchar(32) = NULL,
  @city varchar(255) = NULL,
  @countryCode varchar(2) = NULL,
  @postalCode varchar(5) = NULL
AS
BEGIN

  if @customerId is null
      EXEC AddCustomer
          @email,
          @phoneNumber,
          @customerId output

   if @addressId is null and
      @street is not null and
```

```
      @houseNumber is not null and
      @city is not null and
      @countryCode is not null and
      @postalCode is not null
          EXEC AddAddress
              @street,
              @houseNumber,
              @city,
              @countryCode,
              @postalCode,
              @addressId OUTPUT

   if @addressid is not null
       INSERT INTO IndividualCustomersAddresses(customerid,
addressid)
       VALUES (@customerId, @addressId)

   INSERT INTO IndividualCustomers(CustomerId, FirstName,
LastName) VALUES (@customerId, @firstName, @lastName)
   INSERT INTO LoyaltyCards( CustomerId, DiscountId ) VALUES(
@customerId, (select MAX(DiscountId) from Discounts))
END
```

Procedura AddCompanyCustomer

Procedura służąca do dodania nowego klienta firmowego. Dla nieistniejącego klienta wewnątrz tabeli `Customers`, tworzy dla niego nowy, odpowiedni wpis

```
CREATE OR ALTER PROCEDURE AddCompanyCustomer
  @name varchar(255),
  @NIP varchar(32),
  @customerId INT = NULL OUTPUT ,
  @email varchar(255),
  @phoneNumber varchar(15),
  @addressId INT = NULL,
  @street varchar(255) = NULL,
  @houseNumber varchar(32) = NULL,
  @city varchar(255) = NULL,
  @countryCode varchar(2) = NULL,
  @postalCode varchar(5) = NULL
AS
BEGIN
  IF @customerId is null
      EXEC AddCustomer
          @email,
          @phoneNumber,
          @customerId output

  if @addressId is null and
```

```
      @street is null and
      @houseNumber is null and
      @city is null and
      @countryCode is null and
      @postalCode is null
       THROW 51000, 'Invalid arguments, pass @addressId or pass
data to new specific address',1


  if @addressId is null
       EXEC AddAddress
           @street,
           @houseNumber,
           @city,
           @countryCode,
           @postalCode,
           @addressId output

  INSERT INTO CompanyCustomers(CustomerId, CompanyName, NIP,
AddressId) VALUES (@customerId, @name, @NIP, @addressId)
END
```

Procedura AddEmployee

Procedura dodaje informacje o nowo zatrudnionym pracowniku do bazy danych.

```
CREATE PROCEDURE AddEmployee
           (@Firstname varchar(32),
           @Lastname varchar(64),
           @HouseNumber varchar(32),
           @Street varchar(255),
           @City varchar(255),
           @Country varchar(2),
       @PostalCode varchar(255),
           @Avatar varchar(255) = null,
           @Role nvarchar(255),
           @HiredDate datetime)
AS
BEGIN


   BEGIN

  Declare @addressId int = NULL
  EXEC AddAddress
           @street,
           @houseNumber,
```

```
          @city,
          @country,
          @postalCode,
          @addressId output


       INSERT INTO Employees(FirstName, LastName, Avatar, Role,
HiredDate, AddressId)
       VALUES(@Firstname, @Lastname, @Avatar, @Role, @HiredDate,
@addressId)

   END
END
```

Procedura ModifyEmployee

Procedura modyfikująca dane o pracowniku. W zależności od dostarczonych argumentów, procedura uaktualnia podane wartości. Najważniejsze z nich to:

- Ponowne zatrudnienie pracownika (tracimy informacje o poprzednim okresie zatrudnienia korzystając z tej metody, alternatywnie można dodać go jako nowego pracownika korzystając z opcji append)
- Zwolnienie pracownika

```
CREATE PROCEDURE ModifyEmployee
          (@EmployeeId int,
          @LastName varchar(64) = null,
          @HouseNumber varchar(32) = null,
          @Street varchar(255) = null,
          @City varchar(255) = null,
          @Country varchar(2) = null,
          @Avatar varchar(255) = null,
          @Role nvarchar(255) = null,
          @Fire bit = 0,
          @Hire bit = 0)
AS
BEGIN

   IF @LastName IS NOT NULL
   BEGIN
       UPDATE Employees
```

```
        SET LastName = @LastName
        WHERE EmployeeId = @EmployeeId
    END

    IF @HouseNumber IS NOT NULL
    BEGIN
        UPDATE Addresses
        SET HouseNumber = @HouseNumber
        WHERE AddressId = (SELECT AddressId FROM Employees WHERE
EmployeeId = @EmployeeId)
    END

    IF @Street IS NOT NULL
    BEGIN
        UPDATE Addresses
        SET Street = @Street
        WHERE AddressId = (SELECT AddressId FROM Employees WHERE
EmployeeId = @EmployeeId)
    END

    IF @City IS NOT NULL
    BEGIN
        UPDATE Cities
        SET Name = @City
        WHERE CityId = (SELECT CityId FROM Addresses WHERE
AddressId = (SELECT AddressId FROM Employees WHERE EmployeeId =
@EmployeeId))
    END

    IF @Country IS NOT NULL
    BEGIN
        UPDATE Cities
        SET CountryCode = @Country
        WHERE CityId = (SELECT CityId FROM Addresses WHERE
AddressId = (SELECT AddressId FROM Employees WHERE EmployeeId =
@EmployeeId))
    END

    IF @Avatar IS NOT NULL
    BEGIN
        UPDATE Employees
        SET Avatar = @Avatar
        WHERE EmployeeId = @EmployeeId
    END

    IF @Role iS NOT NULL
    BEGIN
        UPDATE Employees
```

```
        SET Role = @Role
        WHERE EmployeeId = @EmployeeId
    END

    IF @Hire = 1 AND @Fire = 1
    BEGIN
        ;
        THROW 51000, 'You try to hire and fire the employee at the
same time!', 1
    END

    ELSE IF @Hire = 1
    BEGIN
        UPDATE Employees
        SET HiredDate = GETDATE(), FiredDate = NULL
    WHERE EmployeeId = @EmployeeId
    END

    ELSE IF @Fire = 1
    BEGIN
        UPDATE Employees
        SET FiredDate = GETDATE()
    WHERE EmployeeId = @EmployeeId
    END
END
```

Procedura AddEmployeeToOrder

Dodaje zamówienie

```
CREATE PROCEDURE addEmployeeToOrder
          (@EmployeeId int,
          @OrderId int)
AS
BEGIN

    IF NOT EXISTS (SELECT * FROM Employees AS E WHERE E.EmployeeId =
@EmployeeId)
    BEGIN
        ;
        THROW 51000, 'This person does not exist in database', 1
    END
    ELSE IF NOT EXISTS (SELECT * FROM Orders WHERE OrderId =
@OrderId)
    BEGIN
        ;
        THROW 51001, 'Such order was never ordered', 1
```

```
    END
    ELSE IF DATEDIFF(hour,  (SELECT OrderDate FROM Orders WHERE
Orderid = @OrderId), GETDATE()) > 12
    BEGIN
        ;
        THROW 51002, 'Cannot modify order employee in orders
ordered more than 12 hours ago!', 1
    END

    ELSE

    BEGIN
        INSERT INTO EmployeesDetails(EmployeeId, OrderId)
        VALUES(@EmployeeId, @OrderId)
    END
END
```

Procedura AddOrder

Funkcja służąca do składania pełnego zamówienia. W celu jej użycia niezbędne jest podanie id istniejącej rezerwacji lub datę odbioru na wynos (wraz z id klienta). W celu zachowania zgodności z zasadami ACID oraz w celach zapewnienia spójności danych, do realizacji tej procedury wykorzystaliśmy transakcje.

```
CREATE OR ALTER PROCEDURE AddOrder
    @reservationId int = null,
    @customerId int = null,
    @orders FullOrder READONLY,
    @invoice bit = 0,
    @isPaid bit = 0,
    @takeawayDate DATETIME = null,
    @orderId INT = null OUTPUT
AS
BEGIN
    DECLARE @ordersCount INT
    SELECT @ordersCount = COUNT(*) FROM @orders

    IF @ordersCount = 0
        throw 51000, 'Order cannot be empty', 1;

    IF NOT EXISTS (SELECT * FROM Reservations WHERE ReservationId =
@reservationId ) and @takeawayDate is null
        throw 51001, 'This reservation does not exists', 1;

    IF @customerId is not null and NOT EXISTS (SELECT * FROM
Customers WHERE CustomerId = @customerId)
        THROW 51002, 'Customer does not exists', 1
```

```
   IF @customerId is null and @reservationId is not null
      SELECT @customerId = CustomerId
      FROM Reservations
      WHERE ReservationId = @reservationid

   IF @customerId is null and @reservationId is null
       THROW 51003, 'Specify reservationId or customer', 1

   IF @reservationId is not null and @takeawayDate is not null
       THROW 51004, 'You cannot reserve table and takeaway order
in one moment', 1

   BEGIN TRANSACTION;

   BEGIN TRY
       PRINT 'New order transaction begin'

       --- CREATE NEW PARENT ORDER ---

       INSERT INTO Orders( CustomerId, Invoice, IsPaid, OrderDate,
ReservationId)
       VALUES (@customerId, @invoice, @isPaid, GETDATE(),
@reservationId)

       SET @orderId = SCOPE_IDENTITY()

       --- CREATE OPTIONAL TAKEAWAY ORDER ---

       IF @takeawayDate is not null
           INSERT INTO TakeawayOrders(OrderId, TakeawayDate)
           VALUES (@orderId, @takeawayDate)

       --- ADD EACH MEAL ELEMENTS ---

       DECLARE @mealIdToHandle INT
       DECLARE @quantityToHandle INT
       DECLARE @lastMealIdToHandle INT

       SELECT TOP 1
           @lastMealIdToHandle = mealid,
           @quantityToHandle = quantity
       from @orders
       ORDER BY mealid

       SET @mealIdToHandle = @lastMealIdToHandle

       WHILE @mealIdToHandle IS NOT NULL
```

```
        BEGIN

            IF NOT EXISTS (SELECT * FROM MenuView WHERE MenuMealId =
@mealIdToHandle)
                THROW 51009, 'Meal not found', 1

            DECLARE @available INT
            SELECT @available = Avaliable
            FROM MenuView
            WHERE MenuMealId = @mealIdToHandle

            IF @available < @quantityToHandle
                THROW 51001, 'Not enough available dishes', 1

            EXEC AddOrderDetail @Orderid = @orderId, @Quantity =
@quantityToHandle, @MenuMealId = @mealIdToHandle

            PRINT 'Ordered: ' + cast( @mealIdToHandle as VARCHAR) +
' | quantity: ' + cast( @quantityToHandle as VARCHAR)

            SET @lastMealIdToHandle = @mealIdToHandle
            SET @mealIdToHandle = null
            SET @quantityToHandle = null

            SELECT TOP 1
                @mealIdToHandle = mealid,
                @quantityToHandle = quantity
            FROM @orders
            WHERE MealId > @lastMealIdToHandle
            ORDER BY mealid
        END

        IF @isPaid = 1
            EXEC PayForOrder @OrderId = @orderId, @isPaid = @isPaid

        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        PRINT 'Transaction failded'
        ROLLBACK TRANSACTION;

        DECLARE @ErrorNumber INT
        DECLARE @ErrorState INT
        DECLARE @ErrorMessage VARCHAR(MAX)

        SET @ErrorNumber =IIF( ERROR_NUMBER() < 50000,
ERROR_NUMBER() + 50000, ERROR_NUMBER() );
        SET @ErrorMessage = ERROR_MESSAGE();
```

```
        SET @ErrorState = ERROR_STATE();

        PRINT CONCAT(@ErrorNumber,',', @ErrorMessage,',',
@ErrorState);
        THROW @ErrorNumber, @ErrorMessage, @ErrorState
    END CATCH

    SELECT * FROM OrderDetails
    WHERE OrderId = @orderId
END
```

Procedura PayForOrder

Funkcja stworzona w celu zmiany statusu zamówienia na opłacny.

```
CREATE OR ALTER PROCEDURE PayForOrder @OrderId int,
                              @IsPaid bit
AS
BEGIN
    SET NOCOUNT ON
    BEGIN TRY

        IF @IsPaid != 1
            BEGIN;
                THROW 52000, N'Nieprawidłowe wartośc wejściowa.', 1
            END

        IF NOT EXISTS(SELECT * FROM Orders as O WHERE O.OrderId =
@OrderId)
            BEGIN;
                THROW 52000, N'Nieprawidłowe id zamówienia.', 1
            END

        UPDATE Orders
        SET IsPaid = @IsPaid
        WHERE Orders.OrderId=@OrderId

        DECLARE @cstId int = (SELECT TOP 1 O.CustomerId from Orders
as O WHERE O.OrderId = @OrderId)
        IF EXISTS (SELECT * FROM IndividualCustomers WHERE
CustomerId = @cstId)
            Exec CheckCustomerDiscounts @customerId = @cstId

    END TRY
    BEGIN CATCH
```

```
        DECLARE @msg nvarchar(2048)
            =N'Błąd wykonywania procedury.' + ERROR_MESSAGE();
        THROW 52000, @msg, 1
    END CATCH
END
```

Procedura AddOrderDetail

Procedura ma za zadanie dodać szczegół zamówienia tzn informacje o produkcie i ilości produktu który został zamówiony przy konkretnym zamówieniu.

```
CREATE PROCEDURE AddOrderDetail @OrderId int,
                                @Quantity int,
                                @MenuMealId int
AS
BEGIN
    SET NOCOUNT ON
    BEGIN TRY
        IF NOT EXISTS(SELECT * FROM Orders as O WHERE O.OrderId =
@OrderId)
            BEGIN
                THROW 52000, N'Nieprawidłowe id zamówienia.', 1
            END

        IF @Quantity < 0
            BEGIN
                THROW 52000, N'Nieprawidłowa liczba zamówionych
przedmiotów', 1
            END

        select dbo.GetMenuByDate(GETDATE()) as Menu

        IF @MenuMealId NOT IN (SELECT x.MenuMealId FROM
dbo.GetMenuByDate(GETDATE()) as x)
            BEGIN
                THROW 52000, N'Nieprawidłowa liczba zamówionych
przedmiotów', 1
            END

        INSERT INTO OrderDetails(OrderId, MenuMealId, Quantity)
        VALUES (@OrderId,@MenuMealId,@Quantity)
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)
            =N'Błąd wykonywania procedury.' + ERROR_MESSAGE();
        THROW 52000, @msg, 1
    END CATCH
END
```

Procedura ModifyOrderDetail

Procedura mo modyfikować liczbę zamówionych jednostek produktu.

```
CREATE PROCEDURE ModifyOrderDetail @Quantity int,
                                   @OrderDetailId int
AS
BEGIN
   SET NOCOUNT ON
   BEGIN TRY
       IF NOT EXISTS(SELECT * FROM OrderDetails as O WHERE
O.OrderDetailId = @OrderDetailId)
           BEGIN
               THROW 52000, N'Nieprawidłowe id szczegółu
zamówienia.', 1
           END

       IF @Quantity < 0
           BEGIN
               THROW 52000, N'Nieprawidłowa liczba zamówionych
przedmiotów', 1
           END

       UPDATE OrderDetails
       SET Quantity = @Quantity
       WHERE OrderDetails.OrderDetailId = @OrderDetailId
   END TRY
   BEGIN CATCH
       DECLARE @msg nvarchar(2048)
           =N'Błąd wykonywania procedury.'  + ERROR_MESSAGE();
       THROW 52000, @msg, 1
   END CATCH
END
go
```

Procedura AddMeal

Celem procedury jest dodanie nowego posiłku z wybraną kategorią.

```
CREATE OR ALTER PROCEDURE AddMeal
   @name varchar(255),
   @category varchar(255),
   @isPremium bit,
   @mealId INT = NULL OUTPUT
```

```
AS
BEGIN
   DECLARE @categoryId int = NULL

   SELECT @categoryId = CategoryId
   from Categories
   WHERE Name=@category

   IF @categoryId is null
   begin
       THROW 51000, 'Invalid category name',1
   end

   INSERT INTO Meals( CategoryId, Name, IsPremium) VALUES
(@categoryId, @name, @isPremium)

   SET @mealId = SCOPE_IDENTITY()
END
```

Procedura AddMealToMenu

Celem procedury jest dodanie nowego wpisu do (domyślnie) aktualnego Menu.

```
CREATE OR ALTER PROCEDURE AddMealToMenu
   @mealId int,
   @price int = 0,
   @quantity int = 0,
   @startDate datetime = null,
   @endDate datetime = null,
   @menuMealId int = null output
AS
BEGIN
   IF @startDate is null
       SET @startDate = GETDATE()

   INSERT INTO Menu(MealId, StartDate, EndDate, Price, Quantity)
VALUES (@mealId, @startDate, @endDate, @price, @quantity)
   SET @menuMealId = SCOPE_IDENTITY()
END
```

Procedura RemoveMealFromMenu

Procedura służy do usunięcia posiłku z aktualnego menu. Procedura pozwala na sprecyzowanie endDate danego posiłku (domyślnie jest to aktualny czas). Dodatkowo możemy wybrać, czy chcemy usunąć tylko konkretny wpis po MenuMealId lub też po MealId

```
CREATE OR ALTER PROCEDURE RemoveMealFromMenu
```

```
    @mealId int = NULL,
    @menuMealId int = null,
    @endDate datetime = null
AS
BEGIN

    IF @mealId is null and @menuMealId is null
        THROW 52000, 'Invalid arguments, at least one of @mealId or
@menuMealId must be passed', 1

    IF @endDate is null
        SET @endDate = GETDATE()

    IF @menuMealId is not null
        UPDATE Menu
        SET EndDate=@endDate
        WHERE MenuMealId=@menuMealId
    else
        UPDATE Menu
        SET EndDate=@endDate
        WHERE MealId = @mealId
            and (
                EndDate is null OR
                EndDate < GETDATE()
            )
END
```

Procedura AddCategory

Celem procedury jest dodanie nowej kategori produktów.

```
CREATE PROCEDURE AddCategory @CategoryName varchar(255)
AS
BEGIN
    SET NOCOUNT ON
    BEGIN TRY
        INSERT INTO Categories(Name)
        values (@CategoryName)
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)
            =N'Błąd wykonywania procedury.'  + ERROR_MESSAGE();
        THROW 52000, @msg, 1
    END CATCH
END
go
```

Procedura AddCompanyReservation

Procedura dodaje rezerwację dla firmy. W zależności, czy podamy listę gości, czy nie - tworzy rezerwację dla firmy lub dodatkowo zapisuję liste osób, dla których została stworzona rezerwacja.

```
CREATE PROCEDURE [dbo].[AddCompanyReservation] (@CustomerId int,
@Seats int, @StartDate datetime, @Guests GuestList READONLY)
AS
BEGIN
  -- Klient firmowy musi istnieć!
  IF not exists (select Customerid from CompanyCustomers where
CustomerId = @CustomerId)
  BEGIN
     ;
     THROW 51000, 'Such Company Cusotmer does not exist!', 1
  END

  -- rezerwacja dla co najmniej 2 osób
  IF @Seats < 2
  BEGIN
     ;
     THROW 51000, 'Too few seats!', 1
  END


  -- data rezerwacji musi być późniejsza niż data teraźniejsza
(ustalamy, że różnica abs(getdate() - startDate) > 15 minut

  IF DATEDIFF(minute, getdate(), @startDate) < 15
  BEGIN
     ;
     THROW 51000, 'You cannot make reservation less than 15
minutes before it!', 1
  END



  Insert into Reservations (CustomerId, Seats, StartDate, EndDate,
Status)
  values(@CustomerId, @Seats, @StartDate,
DATEADD(hour,2,@StartDate), 'Awaiting')
  DECLARE @ReservationId int = SCOPE_IDENTITY()

  IF EXISTS (select CustomerId from @Guests)
  BEGIN
```

```
    INSERT INTO CompanyCustomerList (ReservationId, CustomerId)
    SELECT @ReservationId, CustomerId FROM @Guests
  END



END

GO
```

Procedura AddIndividualReservation

Procedura tworzy sprawdza, czy możliwe jest utworzenie rezerwacji na podstawie dostarczonego jako argument orderID. Natępnie tworzy rezerwację, dodaje reservationID do zamówienia oraz zajmuje odpowiednią liczbę stolików.

```
ALTER   PROCEDURE [dbo].[AddIndividualReservation] @CustomerId int
, @Products FullOrder READONLY, @Seats int, @StartDate datetime,
@Invoice int, @isPaid int AS
BEGIN

  -- osoba musi istnieć
  IF not exists (select Customerid from IndividualCustomers where
CustomerId = @CustomerId)
  BEGIN
    ;
    THROW 51000, 'Such Cusotmer does not exist!', 1
  END

  -- rezerwacja dla co najmniej 2 osób
  IF @Seats < 2
  BEGIN
    ;
    THROW 51000, 'Too few seats!', 1
  END


  -- data rezerwacji musi być późniejsza niż data teraźniejsza
(ustalamy, że różnica abs(getdate() - startDate) > 15 minut

  IF DATEDIFF(minute, getdate(), @startDate) < 15
  BEGIN
    ;
    THROW 51000, 'You cannot make reservation less than 15
minutes before it!', 1
  END
```

```
  -- owoce morza obsługuje trigger (swoją drogą do poprawy)


  -- zaczynamy od sprawdzenia, czy produkty w podanej ilości w ogóle są dostępne
  -- sprawdzamy czy ilość elementów po zrobieniu joina będzie taka sama jak przed joinem (w join warunek where available > quantity) i odrzucamy jeśli zamówiono przynajmniej jeden produkt niedostępny

  declare @Check int = (select count(*) from MenuView inner join Menu as MN on Mn.MenuMealId = MenuView.MenuMealId inner join @products as p on p.mealId = MN.MenuMealId)
  IF @Check <> (select count(*) from @Products)
  BEGIN
     ;
     THROW 51000, 'One or more of the ordered products is not available', 1
  END

  -- produkty są dostępne, więc sprawdzamy WZ i WK - kolejność raczej bez znaczenia
  declare @WK int = (select count(*) from Orders where CustomerId = @CustomerId)
  IF @WK < (select Wk from ReservationVariables)
  BEGIN
     ;
     THROW 51000, 'Customer has not made orders enough times', 1
  END


  -- tu wersja bez rabatów
  --declare @WZ int = (select sum(Mv.Price * P.quantity) from MenuView as Mv inner join @Products as P on p.mealId = Mv.MenuMealId)
  --IF @WZ < (select WZ from ReservationVariables)
  --BEGIN

  -- ;
     --THROW 51000, 'Order value is to low', 1
  --END

  -- ###################### wersja z rabatami ########################

  declare @R2 float = (
     select R2 from Onetimediscounts
     inner join BigDiscounts as b on b.BigDiscountId = OneTimeDiscounts.BigDiscountId
```

```
      where getdate() between ActivationDate and ExpireDate and
orderid is null and OneTimeDiscounts.CustomerId = @CustomerId
   )



   declare @R1 float = (
      select R1 from LoyaltyCards inner join Discounts as d on
d.DiscountId = LoyaltyCards.DiscountId where
loyaltycards.ActivationDate is not null and CustomerId =
@CustomerId
   )

   declare @R float

   IF ISNULL(@R2, 0) >  ISNULL(@R1, 0)
   BEGIN
      SET @R = @R2
   END
   ELSE
   BEGIN
      SET @R = @R1
   END

   declare @WZ float
   SET @WZ = (select sum(Mv.Price * P.quantity * (1- @R)) from
MenuView as Mv inner join @Products as P on p.mealId =
Mv.MenuMealId)

   IF @WZ < (select WZ from ReservationVariables)
   BEGIN

      ;
      THROW 51000, @WZ, 1
   END

   -- ############################ koniec wersji z discountami
####################


   -- WZ I WK są spełnione, w dodatku w menu jest wystarczająco dużo
posiłków, więc można rezerwować


   PRINT @WZ

   BEGIN TRANSACTION
```

```
  BEGIN TRY
        Insert into Reservations (CustomerId, Seats, StartDate,
EndDate, Status)
        values(@CustomerId, @Seats, @StartDate,
DATEADD(hour,2,@StartDate), 'Awaiting')

        declare @ReservationId int = SCOPE_IDENTITY()

        exec AddOrder @ReservationId, @CustomerId, @Products,
@Invoice, @isPaid

        -- tutaj dodajemy do bazy danych nasze zamówienie i na tym
kończymy (dopiero potwierdzenie stolika przez managera doda do
rezerwacji stolik!

        COMMIT TRANSACTION

  END TRY
  BEGIN CATCH
      PRINT 'Reservation failed'
        ROLLBACK TRANSACTION

        DECLARE @ErrorNumber INT
        DECLARE @ErrorState INT
        DECLARE @ErrorMessage VARCHAR(MAX)

        SET @ErrorNumber =IIF( ERROR_NUMBER() < 50000,
ERROR_NUMBER() + 50000, ERROR_NUMBER() );
        SET @ErrorMessage = ERROR_MESSAGE();
        SET @ErrorState = ERROR_STATE();

        PRINT CONCAT(@ErrorNumber,',', @ErrorMessage,',',
@ErrorState);
        THROW @ErrorNumber, @ErrorMessage, @ErrorState

   END CATCH

END
```

Procedura ChangeReservationStatus

Procedura zmienia status rezerwacji na podany. Zmiana na ten sam status co poprzednio ustawiony nie jest uznawana za błąd.

```
CREATE PROCEDURE [dbo].[ChangeReservationStatus](@ReservationId
int, @Status varchar(255))
AS
```

```
BEGIN
  IF @ReservationId NOT IN (Select ReservationId from Reservations
where ReservationId = @ReservationId)
  BEGIN
     ;
     THROW 51000, 'There is no such reservation!', 1
  END

  IF @Status NOT IN ('Confirmed', 'Rejected', 'Awaiting')
  BEGIN
     ;
     THROW 51000, 'Incorrect status!', 1
  END

  UPDATE Reservations
  SET Status = @Status
  WHERE ReservationId = @ReservationId

END
```

Procedura AddTableToReservation

Procedura dodaje stolik do rezerwacji. Lista stolików jest podawana przez managera / pracownika akceptującego rezerwację.

```
CREATE PROCEDURE [dbo].[AddTableToReservation] (@ReservationId int,
@Tables TableList READONLY)
AS
BEGIN

  DECLARE @Seats int = (select seats from Reservations where
ReservationId = @ReservationId)
  DECLARE @GivenSeats int = (SELECT SUM(ET.Size) FROM EmptyTables
AS ET inner join @Tables as T on T.TableId = ET.TableId)

  IF @GivenSeats < @Seats
  BEGIN
     PRINT @GivenSeats;
     THROW 51000, 'Given tables are to small for this
reservation!', 1
  END


  INSERT INTO ReservationDetails(ReservationId, TableId)
  SELECT @ReservationId, Tableid FROM @Tables
END
GO
```

Procedura AddTakeawayOrder

Procedura ma za zadanie stworzyć informacje o tym że określone zamówienie będzie odbierane przez zamawiającego określonego dnia.

```
CREATE PROCEDURE AddTakeawayOrder @OrderId int,
                                  @TakeawayDate datetime
AS
BEGIN
   SET NOCOUNT ON
   BEGIN TRY
       IF NOT EXISTS(SELECT * FROM Orders as O WHERE O.OrderId =
@OrderId)
           BEGIN
               THROW 52000, N'Nieprawidłowe id zamówienia.', 1
           END

       IF @TakeawayDate < GETDATE()
           BEGIN
               THROW 52000, N'Nieprawidłowa data odbioru', 1
           END

       INSERT INTO TakeawayOrders(OrderId, TakeawayDate)
       values (@OrderId, @TakeawayDate)
   END TRY
   BEGIN CATCH
       DECLARE @msg nvarchar(2048)
           =N'Błąd wykonywania procedury.' ' ' + ERROR_MESSAGE();
       THROW 52000, @msg, 1
   END CATCH
END
go
```

Procedura AddTable

Procedura ma za zadanie dodać nowy stół w restauracji.

```
CREATE PROCEDURE AddTable @Size int
AS
BEGIN
   SET NOCOUNT ON
   BEGIN TRY
```

```
        IF @Size > 0
            BEGIN
                THROW 52000, N'Nieprawidłowa wielkość stołu.', 1
            END

        INSERT INTO Tables(Size)
        values (@Size)
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)5200
            =N'Błąd wykonywania procedury.' + ERROR_MESSAGE();
        THROW 52000, @msg, 1
    END CATCH
END
go
```

Procedura ModifyTable

Procedura ma za zadanie zmienić wielkość stołu w restauracji.

```
CREATE PROCEDURE ModifyTable @Size int,
                             @TableId int
AS
BEGIN
    SET NOCOUNT ON
    BEGIN TRY

        IF @Size > 0
            BEGIN
                THROW 52000, N'Nieprawidłowa wielkość stołu.', 1
            END

        IF NOT EXISTS(SELECT * FROM Tables as T WHERE
T.TableId=@TableId)
            BEGIN;
                THROW 52000, N'Stolik nie istnieje.', 1
            END

        Update Tables
        SET Size = @Size
        WHERE Tables.TableId=@TableId
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)
            =N'Błąd wykonywania procedury.' + ERROR_MESSAGE();
        THROW 52000, @msg, 1
    END CATCH
```

```
END
go
```

Procedura ChangeAddress

Procedura zmienia adres zamieszkania klienta indywidualnego, pracownika restauracji lub w przypadku klienta firmowego - adres siedziby głównej firmy.

```
CREATE PROCEDURE [dbo].[changeAddress] (@AddressID int,
@HouseNumber varchar(32), @Street varchar(255), @CityName
varchar(255), @Country varchar(2), @PostalCode varchar(5))
AS
BEGIN


  IF @AddressID not in (select addressID from Addresses)
     BEGIN
     ;
     THROW 51000, 'nie ma takiego adresu', 1
  END

  ELSE
     BEGIN
     DECLARE @row_exists INT = 0
     SELECT @row_exists = COUNT(*)
     FROM Cities
     WHERE name=@CityName
     AND CountryCode=@country
     AND PostalCode=@postalCode

     DECLARE @CityId int
  -- wiersz nie istnieje
     IF @row_exists = 0
     begin
        INSERT INTO Cities(name, countrycode, postalcode) VALUES
(@Cityname, @country, @postalCode)
        SET @cityid = SCOPE_IDENTITY()
     end
     else
     begin
        SELECT @cityid = CityId
        FROM Cities
        WHERE name=@Cityname
        AND CountryCode=@country
        AND PostalCode=@postalCode
     end
```

```
          UPDATE Addresses
          SET HouseNumber = @HouseNumber, Street = @Street, CityId
= @CityId
           WHERE AddressId = @AddressID

  END
END
GO
```

Procedura AddCity

Funkcja dodająca nowe miasto do tabeli `Cities` i zwracająca `cityid` nowego wpisu. Jeśli już dany wpis występował w danej tabeli, to funkcja zwraca `cityid` znalezionego, pasującego wiersza.

```
CREATE OR ALTER PROCEDURE AddCity
   @name varchar(255),
   @countryCode varchar(2),
   @postalCode varchar(5),
   @cityid INT = NULL OUTPUT
AS
BEGIN
   DECLARE @row_exists INT = 0
   SELECT @row_exists = COUNT(*)
   FROM Cities
   WHERE name=@name
     AND CountryCode=@countryCode
     AND PostalCode=@postalCode

   IF @row_exists = 0
   begin
       INSERT INTO Cities(name, countrycode, postalcode) VALUES
(@name, @countryCode, @postalCode)
       SET @cityid = SCOPE_IDENTITY()
   end
   else
   begin
       SELECT @cityid = CityId
       FROM Cities
       WHERE name=@name
       AND CountryCode=@countryCode
       AND PostalCode=@postalCode
   end
END
```

Procedura AddAddress

Funkcja dodająca nowy address do tabeli `Cities` i zwracająca `cityid` nowego wpisu.

```
CREATE OR ALTER PROCEDURE AddAddress
   @street varchar(255),
   @houseNumber varchar(32),
   @city varchar(255),
   @countryCode varchar(2),
   @postalCode varchar(5),
   @addressId INT = NULL OUTPUT
AS
BEGIN
   DECLARE @cityId INT

   EXEC AddCity @city,@countryCode, @postalCode, @cityId OUTPUT

   INSERT INTO Addresses(cityid, street, housenumber) VALUES
(@cityId, @street, @houseNumber)

   SET @addressId = SCOPE_IDENTITY()
END
```

Procedura AddDiscount

Dodawanie nowej zniżki do przyznawania klientom. Automatycznie wyłącza poprzednią i aktywuje nową.

```
CREATE PROCEDURE AddDiscount @R1 float,
                             @K1 money,
                             @Z1 int
AS
BEGIN
   SET NOCOUNT ON
   BEGIN TRY

       IF @R1 > 0 AND @R1 <= 1
           BEGIN;
               THROW 52000, N'Rabat nie może być wiekszy od 1 i
mnieszy od 0.', 1
           END

       IF @Z1 > 0
           BEGIN;
               THROW 52000, N'Liczba zamówień musi być wieksza od
0.', 1
           END
```

```
        IF @K1 > 0
            BEGIN;
                THROW 52000, N'Kwota zamówień musi być wieksza od
0.', 1
            END


        UPDATE Discounts
        SET ExpireDate = GETDATE()
        WHERE Discounts.ExpireDate IS NULL

        INSERT INTO Discounts(R1, Z1, K1, ActivationDate)
        VALUES (@R1, @Z1, @K1, GETDATE())

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)
            =N'Błąd wykonywania procedury.' + ERROR_MESSAGE();
        THROW 52000, @msg, 1
    END CATCH
END
go
```

Procedura AddBigDiscount

Procedura wprowadza nowe parametry zniżki jednorazowej.

```
CREATE PROCEDURE addBigDiscount(@R2 float, @K2 money, @Duration
datetime)
AS
BEGIN
    INSERT INTO BigDiscounts(R2, K2, Duration)
    VALUES(@R2, @K2, @Duration)
END
GO
```

Procedura ActivateDiscountOnOrder

Procedura służąca do przypisania jednorazowej (Aktywacja) zniżki do zamówienia.

```
CREATE PROCEDURE ActivateDiscountOnOrder
  @customerId INT = NULL,
  @orderId INT = NULL
AS
```

```
BEGIN
  IF @customerId is null or NOT EXISTS(SELECT * FROM Customers AS C
WHERE C.CustomerId = @customerId)
      THROW 52000, 'Invalid argument: customerId', 1

  IF @customerId is null or NOT EXISTS(SELECT * FROM Orders AS O
WHERE O.OrderId = @orderId)
      THROW 52000, 'Invalid argument: orderId', 1

  IF EXISTS(SELECT * FROM OneTimeDiscounts WHERE CustomerId =
@customerId
                                          AND ExpireDate is not
null
                                          AND ExpireDate <
GETDATE())
      begin
             UPDATE OneTimeDiscounts
             SET OrderId = @orderId
             where CustomerId = @CustomerId
      end
   else
       THROW 52000, 'Customer doest have free discount', 1

END
```

Procedura CheckCustomerDiscounts

Procedura służąca do aktywowania zniżek po złożeniu zamówienia.

```
CREATE OR ALTER PROCEDURE CheckCustomerDiscounts @customerId INT =
NULL
AS
BEGIN
  IF @customerId is null or @customerId = 0 or NOT EXISTS(SELECT *
FROM IndividualCustomers AS C WHERE C.CustomerId = @customerId)
   begin
       THROW 52000, 'Invalid argument', 1
   end


   IF EXISTS(SELECT * FROM LoyaltyCards AS LC WHERE CustomerId =
@customerId AND ActivationDate is null)
       begin
```

```
            declare @z1 int = (SELECT top 1 D.Z1
                               FROM LoyaltyCards AS LC
                                        join Discounts D on
D.DiscountId = LC.DiscountId
                               WHERE LC.CustomerId = @CustomerId)

            declare @z1c int = (SELECT COUNT(*)
                                FROM OrdersInfo AS OI
                                         JOIN LoyaltyCards as LC on
LC.CustomerId = OI.CustomerId
                                         JOIN Discounts D2 on
LC.DiscountId = D2.DiscountId
                                WHERE OI.CustomerId = @CustomerId
                                  AND OI.Value > D2.K1)

            if (@z1c > @z1)
                begin
                    UPDATE LoyaltyCards
                    SET ActivationDate = GETDATE()
                    WHERE CustomerId = @customerId
                end

        end


    IF EXISTS(SELECT * FROM OneTimeDiscounts AS LC WHERE CustomerId
= @customerId AND ActivationDate is null)
        begin

            declare @k1 money = (SELECT top 1 D.Z1
                                 FROM LoyaltyCards AS LC
                                          join Discounts D on
D.DiscountId = LC.DiscountId
                                 WHERE LC.CustomerId = @CustomerId)

            declare @k1c money = (SELECT SUM(OI.Value)
                                  FROM OrdersInfo as OI
                                  WHERE CustomerId = @customerId
                                  GROUP BY CustomerId)

            IF @k1 < @k1c
                begin
                    UPDATE OneTimeDiscounts
                    SET ActivationDate = GETDATE()
                    where CustomerId = @CustomerId

                    declare @span int = (SELECT TOP 1
BigDiscounts.Duration
```

```
                                        FROM BigDiscounts
                                                JOIN
OneTimeDiscounts OTD on BigDiscounts.BigDiscountId =
OTD.BigDiscountId
                                        WHERE CustomerId =
@CustomerId)

                    UPDATE OneTimeDiscounts
                    SET ExpireDate = DATEADD(day, @span, GETDATE())
                    where CustomerId = @customerId

                end

        end

END
```

Procedura RefreshMenu

Procedura służąca do odświeżania zawartości menu, wywoływana raz na 2 tygodnie. Każde jej użycie wiąże się z wymianą 10 posiłków z menu. Najpierw do wymiany są typowane dania, które są już przynajmniej od 2 tygodnii w menu, następnie jeśli nie udało się znaleźć 10 takich, to wybierani są losowo kolejni kandydaci.
Gdy osiągniemy liczbę 10 kandydatów, następuje wymiana menu.
Procedura pomija produkty z kategorii “owoce morza”
Uwaga: cenę oraz ilość każdego nowego elementu menu musi ustalić manager, domyślnie są one ustawione na 0 (w tym celu manager wykorzystuje procedurę SetMenuMealProperty).

```
CREATE OR ALTER PROCEDURE RefreshMenu
   @date DATETIME = NULL
AS
BEGIN

   IF @date is null
       SET @date = CONVERT(date, DATEADD(day, 1, getdate()))

   --- GET CURRENT MENU ---

   DECLARE @sharedMeals TABLE (MealId int);

   INSERT INTO @sharedMeals (MealId) SELECT TOP 10 MealId FROM (
       SELECT M.MealId FROM dbo.GetMenuByDate(@date) AS M
```

```
        INTERSECT
        SELECT M.MealId FROM dbo.GetMenuByDate(DATEADD(week , -2,
@date)) AS M
        INTERSECT
        SELECT M.MealId FROM Meals M WHERE dbo.isSeafood ( M.MealId
) = 0
    ) T

    DECLARE @sharedMealsCount int;
    SELECT  @sharedMealsCount = COUNT(*) FROM @sharedMeals

    IF @sharedMealsCount < 10
        INSERT INTO @sharedMeals
        SELECT TOP (10 - @sharedMealsCount) M.MealId
        FROM dbo.GetMenuByDate(@date) M
        WHERE NOT EXISTS (
            SELECT *
            FROM @sharedMeals
            WHERE MealId = M.MealId
            ) AND dbo.isSeafood (M.MealId) = 0
        ORDER BY NEWID()

    --- GENERATE NEW MEALS ---

    DECLARE @newMeals TABLE (MealId Int);

    INSERT INTO @newMeals (MealId)
    SELECT TOP 10 MealId
    FROM Meals
    WHERE dbo.isSeafood (MealId) = 0 and MealId not in (
        SELECT MealId from dbo.GetMenuByDate (@date)
    )
    ORDER BY NEWID()

    select * from @newMeals

    -- REMOVE OLD MENU ---

    DECLARE @mealIdToHandle INT
    DECLARE @lastMealIdToHandle INT

    SELECT TOP 1 @lastMealIdToHandle = mealid
    from @sharedMeals
    ORDER BY mealid

    SET @mealIdToHandle = @lastMealIdToHandle

    WHILE @mealIdToHandle IS NOT NULL
```

```
    BEGIN
        EXEC RemoveMealFromMenu @mealId = @mealIdToHandle, @endDate
= @date

        PRINT 'Removed: ' + cast( @mealIdToHandle as VARCHAR)

        SET @lastMealIdToHandle = @mealIdToHandle
        SET @mealIdToHandle = null

        SELECT TOP 1 @mealIdToHandle = mealid
        FROM @sharedMeals
        WHERE MealId > @lastMealIdToHandle
        ORDER BY mealid
    END

    --- ADD NEW MEALS ---

    SELECT TOP 1 @lastMealIdToHandle = mealid
    from @newMeals
    ORDER BY mealid

    SET @mealIdToHandle = @lastMealIdToHandle

    SELECT TOP 1 mealid
        FROM @newMeals
        WHERE MealId > @lastMealIdToHandle
        ORDER BY mealid

    WHILE @mealIdToHandle IS NOT NULL
    BEGIN
        EXEC AddMealToMenu @mealId = @mealIdToHandle, @startDate =
@date

        PRINT 'Added ' + cast( @mealIdToHandle as VARCHAR)
        SET @lastMealIdToHandle = @mealIdToHandle
        SET @mealIdToHandle = null

        SELECT TOP 1 @mealIdToHandle = mealid
        FROM @newMeals
        WHERE MealId > @lastMealIdToHandle
        ORDER BY mealid
    END

    SELECT * from MenuView
END
```

Procedura AcceptReservation

Procedura służąca do zmiany statusu rezerwacji na ‘Confirmed’ a także przypisania stolików do danej rezerwacji.

```
CREATE PROCEDURE [dbo].[AcceptReservation] (@ReservationId int,
@Tables TableList READONLY)
AS
BEGIN
   IF @ReservationId NOT IN (SELECT ReservationId from
Reservations)
   BEGIN
         ;
         THROW 51000, 'There is no such ReservationId', 1
   END



  EXEC AddTableToReservation @ReservationId, @Tables



      EXEC ChangeReservationStatus @ReservationId, 'Confirmed'

END
GO
```

Procedura RejectReservation

Procedura służąca do zmiany statusu rezerwacji na ‘Rejected’.

```
CREATE PROCEDURE [dbo].[RejectReservation] (@ReservationId int)
AS
BEGIN
  EXEC ChangeReservationStatus @ReservationId, 'Rejected'
END
GO
```

Procedura SetMenuMealProperty

Procedura służąca do zmiany ceny oraz aktualnej ilości konkretnego elementu menu.

```
CREATE OR ALTER PROCEDURE SetMenuMealProperty
   @menuMealId INT,
   @price MONEY = null,
   @quantity INT = null
AS BEGIN
   UPDATE Menu
```

```
    SET
        Price = @price,
        Quantity = @quantity
    WHERE MenuMealId = @menuMealId
END
```

Funkcje

Funkcja GetEmployees

Funkcja zwraca podstawowe informacje o pracownikach

```
CREATE FUNCTION getEmployees()
RETURNS TABLE
AS

RETURN (SELECT
  E.EmployeeId AS ID,
  E.FirstName + ' ' + E.LastName AS "Imię i nazwisko",
  A.Street + ' ' + A.HouseNumber + ', ' +  C.Name AS "Adres
zamieszkania",
  E.Avatar AS AvatarURL,
  IIF(E.FiredDate IS NULL, E.Role, NULL) AS Stanowisko,
  IIF(E.FiredDate IS NULL, 'Zatrudniony', 'Niezatrudniony') AS
"Stan zatrudnienia"
FROM Employees AS E
INNER JOIN Addresses AS A ON E.AddressId = A.AddressId
INNER JOIN Cities AS C ON A.CityId = C.CityId)
```

Funkcja GetFiredEmployees

Funkcja zwraca listę pracowników zwolnionych

```
CREATE FUNCTION getFiredEmployees()
RETURNS TABLE
AS

RETURN (SELECT
  E.EmployeeId AS ID,
  E.FirstName + ' ' + E.LastName AS "Imię i nazwisko",
  E.HiredDate AS "Data zatrudnienia",
  E.FiredDate AS "Data zwolnienia",
  DATEDIFF(week, E.HiredDate, E.FiredDate) AS "Okres zatrudnienia",
  A.Street + ' ' + A.HouseNumber + ', ' +  C.Name AS "Adres
zamieszkania",
  E.Avatar AS AvatarURL,
  E.Role AS "Stanowisko"


FROM Employees AS E
INNER JOIN Addresses AS A ON E.AddressId = A.AddressId
INNER JOIN Cities AS C ON A.CityId = C.CityId

WHERE E.FiredDate IS NOT NULL)
```

Funkcja GetHiredEmployees

Zwraca listę pracowników aktualnie zatrudnionych.

```
CREATE FUNCTION getHiredEmployees()
RETURNS TABLE
AS

RETURN (SELECT
  E.EmployeeId AS ID,
  E.FirstName + ' ' + E.LastName AS "Imię i nazwisko",
  E.HiredDate AS "Data zatrudnienia",
  A.Street + ' ' + A.HouseNumber + ', ' +  C.Name AS "Adres
zamieszkania",
  E.Avatar AS AvatarURL,
  E.Role AS "Stanowisko"


FROM Employees AS E
INNER JOIN Addresses AS A ON E.AddressId = A.AddressId
INNER JOIN Cities AS C ON A.CityId = C.CityId

WHERE E.FiredDate IS NULL)
```

Funkcja GetIndividualCustomers

Funkcja zwraca podstawowe informacje o wszystkich klientach indywidualnych.

```
CREATE FUNCTION GetIndividualCustomers()
   RETURNS TABLE AS
       RETURN SELECT C.CustomerId AS CustomerId,
                     C.Email as Email,
                     C.PhoneNumber AS Number,
                     IC.FirstName + ' ' + IC.LastName AS Name,
                     C.CreationDate AS CreatedAt,
                     a.HouseNumber + ' ' + a.Street + ' ' + cc.Name + ' ' + cc.PostalCode
as Address
              FROM Customers AS C
                       JOIN IndividualCustomers AS IC on C.CustomerId = IC.CustomerId
                       LEFT JOIN IndividualCustomersAddresses as ICA ON ICA.CustomerId =
C.CustomerId
                       LEFT JOIN Addresses A on ICA.AddressId = A.AddressId
                       LEFT JOIN Cities AS CC ON CC.CityId = A.CityId
```

Funkcja GetCompanyCoustomers

Funkcja zwraca podstawowe informacje o klientach firmowych.

```
CREATE FUNCTION GetCompanyCustomers()
   RETURNS TABLE AS
       RETURN SELECT C.CustomerId AS CustomerId,
                     C.Email as Email,
                     C.PhoneNumber AS PhoneNumber,
                     IC.CompanyName AS CompanyName,
                     IC.NIP AS NIP,
                     C.CreationDate AS CreatedAt,
                     a.HouseNumber + ' ' + a.Street + ' ' + C2.Name + ' ' + C2.PostalCode
as Address
              FROM Customers AS C
                       JOIN CompanyCustomers AS IC on C.CustomerId = IC.CustomerId
                       JOIN Addresses A on IC.AddressId = A.AddressId
                       JOIN Cities C2 on C2.CityId = A.CityId
```

Funkcja GetCoustomerById

Funkcja zwraca podstawowe informacje o kliencie wybranym po Id.

```
CREATE FUNCTION GetCustomerById(@id int)
   RETURNS @Customer TABLE
                     (
                         CustomerId    int,
                         Name          varchar(510),
                         Email         varchar(255),
                         Number        varchar(15),
                         Address       varchar(255),
                         AdditonalData varchar(255),
                         CreatedAt     datetime
                     )
AS
BEGIN
   IF @id IN (SELECT CC.CustomerId FROM CompanyCustomers AS CC)
       INSERT INTO @Customer
       SELECT CC.CustomerId,
              CC.CompanyName,
              CC.Email,
              CC.PhoneNumber,
              CC.Address,
              '{"Nip":"' + CC.NIP + '"}',
              C.CreationDate
       FROM GetCompanyCustomers() as CC
           JOIN Customers AS C ON C.CustomerId = CC.CustomerId
       WHERE CC.CustomerId = @id

   ELSE
```

```
       INSERT INTO @Customer
       SELECT IC.CustomerId,
              IC.Name,
              IC.Email,
              IC.Number,
              IC.Address,
              '{}',
              C.CreationDate
       From GetIndividualCustomers() AS IC
           JOIN Customers AS C ON C.CustomerId = IC.CustomerId
       WHERE IC.CustomerId = @id

   RETURN
end
```

Funkcja GetOrderInformations

Zwraca informacje o konkretnym zamówieniu.

```
CREATE FUNCTION GetOrderInformations(@id int)
   RETURNS @Order TABLE
                  (
                      OrderId    int,
                      CustomerId int,
                      CostWithDiscount      money,
                      Discount   float,
                      IsPaid     bit
                  )
AS
BEGIN
   IF NOT EXISTS(SELECT * FROM Orders AS O WHERE O.OrderId = @id)
       RETURN

   Declare @OrderDate datetime = (SELECT O.OrderDate FROM Orders
AS O WHERE O.OrderId = @id)

   IF EXISTS(SELECT * FROM OneTimeDiscounts AS OTD WHERE
OTD.OrderId = @id)
       INSERT INTO @Order
       SELECT O.OrderId,
              O.CustomerId,
              ISNULL(SUM(OD.Quantity * M.Price * (1 - BD.R2)), 0),
              BD.R2,
              O.IsPaid
       FROM Orders as O
                JOIN OrderDetails OD on O.OrderId = OD.OrderId
                JOIN Menu as M on M.MenuMealId = OD.MenuMealId
```

```
                JOIN OneTimeDiscounts D on O.OrderId = D.OrderId
                JOIN BigDiscounts BD on D.BigDiscountId =
BD.BigDiscountId
        WHERE O.OrderId = @id
        GROUP BY O.OrderId, O.CustomerId, O.IsPaid, BD.R2
    ELSE
        IF (SELECT LC.ActivationDate
                                FROM Orders AS O
                                        JOIN LoyaltyCards LC on
O.CustomerId = LC.CustomerId
                                WHERE O.OrderId = @id) is NULL or
@OrderDate < ISNULL((SELECT LC.ActivationDate
                                FROM Orders AS O
                                        JOIN LoyaltyCards LC on
O.CustomerId = LC.CustomerId
                                WHERE O.OrderId = @id), GETDATE())
            INSERT INTO @Order
            SELECT O.OrderId,
                   O.CustomerId,
                   ISNULL(SUM(OD.Quantity * M.Price), 0),
                   0,
                   O.IsPaid
            FROM Orders as O
                     JOIN OrderDetails OD on O.OrderId = OD.OrderId
                     JOIN Menu as M on M.MenuMealId = OD.MenuMealId
            WHERE O.OrderId = @id
            GROUP BY O.OrderId, O.CustomerId, O.IsPaid
        ELSE
            INSERT INTO @Order
            SELECT O.OrderId,
                   O.CustomerId,
                   ISNULL(SUM(OD.Quantity * M.Price * (1 - D2.R1)),
0),
                   D2.R1,
                   O.IsPaid
            FROM Orders as O
                     JOIN OrderDetails OD on O.OrderId = OD.OrderId
                     JOIN Menu as M on M.MenuMealId = OD.MenuMealId
                     JOIN LoyaltyCards L on O.CustomerId =
L.CustomerId
                     JOIN Discounts D2 on D2.DiscountId =
L.DiscountId
            WHERE O.OrderId = @id
            GROUP BY O.OrderId, O.CustomerId, O.IsPaid, D2.R1

            RETURN
END
go
```

Funkcja GetValueOfOrdersInDay

Funkcja zwraca łączną wartość zamówień danego dnia (włączając zamówienia nieopłacone)

```
CREATE FUNCTION getValueOfOrderInDay(@day int, @month int, @year
int)
RETURNS float
AS
BEGIN

DECLARE @OrderToSum TABLE
(orderVal float)


INSERT INTO @OrderToSum
select OI.Value from OrdersInfo AS OI
WHERE day(OI.OrderDate) = @day AND
     MONTH(OI.OrderDate) = @month AND
     YEAR(OI.OrderDate) = @year

RETURN (Select SUM(orderVal) from @OrderToSum)

END
go
```

Funkcja GetValueOfOrdersInMonth

Funkcja zwraca łączną wartość zamówień danego miesiąca (włączając zamówienia nieopłacone)

```
CREATE FUNCTION getValueOfOrderInMonth(@month int, @year int)
RETURNS float
AS
BEGIN
DECLARE @OrderToSum TABLE
(orderVal float)

INSERT INTO @OrderToSum
select Sum(OI.Value) from OrdersInfo AS OI
WHERE MONTH(OI.OrderDate) = @month AND
     YEAR(OI.OrderDate) = @year

RETURN isnull((Select SUM(orderVal) from @OrderToSum), 0)
END
```

Funkcja GetXBestMeals

Zwraca X najlepszych potraw (najczęściej zamawianych) w historii całego menu(od powstania restuaracji).

```
CREATE FUNCTION GetXBestMeals(@x int)
   RETURNS TABLE AS
       RETURN SELECT TOP (@x) M.MealId          AS MealId,
                              M.Name            AS [Meal name],
                              SUM(OD.Quantity) AS [Ordered Units]
              FROM Meals as M
                       LEFT JOIN Menu Me on M.MealId = Me.MealId
                       LEFT JOIN OrderDetails OD on Me.MenuMealId = OD.MenuMealId
              GROUP BY M.MealId, M.Name
              ORDER BY [Ordered Units] desc
```

Funkcja GetMenuByDate

Zwraca menu danego dnia.

```
CREATE FUNCTION GetMenuByDate(@date datetime)
   RETURNS TABLE AS
       RETURN SELECT DISTINCT M.MealId,
                              M.MenuMealId,
                              Me.Name,
                              M.Price,
                              M.StartDate,
                              M.EndDate
              FROM Menu AS M
                       JOIN Meals Me on Me.MealId = M.MealId
              WHERE M.StartDate < @date
                AND (M.EndDate IS NULL OR
                     M.EndDate > @date)
```

Funkcja GetMealsSoldAtLeastXTimes

Zwraca potrawy sprzedane przynajmniej X razy z aktualnego menu.

```
CREATE FUNCTION GetMealsSoldAtLeastXTimes(@x int)
   RETURNS TABLE AS
       RETURN SELECT M2.MealId,
                     M2.Name,
                     SUM(OD.Quantity) AS UnitsSold
              FROM Menu AS M
                       JOIN Meals M2 on M2.MealId = M.MealId
                       JOIN OrderDetails OD on M.MenuMealId =
OD.MenuMealId
```

```
            WHERE M.EndDate IS NULL
            GROUP BY M2.MealId, M2.Name
            HAVING SUM(OD.Quantity) > @x
```

Funkcja IsTodaysMenuCorrect

Zwraca czy dzisiejsze menu jest poprawne wymienione czy potrzebne są zmiany.
(Zakładamy że w menu jest 20 dań zawsze)

```
CREATE FUNCTION IsTodaysMenuCorrect()
   RETURNS BIT AS
       BEGIN

           DECLARE @count int =
               (SELECT COUNT(*) FROM (SELECT M.MealId FROM
dbo.GetMenuByDate(GETDATE()) AS M
               INTERSECT
               SELECT M.MealId FROM dbo.GetMenuByDate(DATEADD(week
, -2, GETDATE())) AS M) as A)

       RETURN @count <= 10

       end
```

Funkcja IsMenuCorrectByDate

Zwraca czy menu było jest poprawne dla określonej daty.
(Zakładamy że w menu jest 20 dań zawsze).

```
CREATE OR ALTER FUNCTION IsMenuCorrectByDate(@date datetime)
  RETURNS BIT AS
      BEGIN
          DECLARE @count int =
              (SELECT COUNT(*) FROM (SELECT M.MenuMealId FROM
dbo.GetMenuByDate(@date) AS M
              INTERSECT
              SELECT M.MenuMealId FROM
dbo.GetMenuByDate(DATEADD(week, -2, @date)) AS M
              INTERSECT
              SELECT M.MealId FROM Meals M WHERE dbo.isSeafood
(M.MealId) = 0
           ) as A)

       DECLARE @response bit = 0
```

```
        IF @count <= 10
            set @response = 1

    RETURN @response
END
```

Funkcja GetClientsOrderedMoreThanXTimes

Funkcja zwraca listę klientów, którzy złożyli zamówienie więcej niż x razy.

```
CREATE FUNCTION getClientsOrderedMoreThanXTimes(@x int)
RETURNS TABLE
AS
RETURN select CustomerId FROM Orders  Group By CustomerId HAving
Count(OrderId) > @x
```

Funkcja GetEmployeeAddress

Zwraca adres pracownika.

```
CREATE FUNCTION GetEmployeeAddress(@id int)
   RETURNS TABLE AS RETURN
       SELECT e.EmployeeId,
              C.Name,
              A.Street,
              A.HouseNumber
       FROM Employees AS E
                JOIN Addresses A on A.AddressId = E.AddressId
                JOIN Cities C on C.CityId = A.CityId
       WHERE E.EmployeeId = @id
```

Funkcja isSeafood

Funkcja zwraca informacje, czy dany posiłek jest owocem morza.

```
CREATE OR ALTER FUNCTION isSeafood( @mealId int )
RETURNS BIT
AS BEGIN
   DECLARE @category VARCHAR(max)

   SELECT @category=Categories.name FROM Meals
   INNER JOIN Categories
   ON Meals.CategoryId = Categories.CategoryId
   WHERE Meals.MealId = @mealId
```

```
    return IIF( @category = 'Seafood', 1, 0)
END
```

Triggery

Trigger DeleteOrderOnReservationRejection

Trigger ma automatycznie usuwa zamówienie wraz z szczegółami zamówienia, jeżeli rezerwacja zostanie odrzucona.

```
CREATE OR ALTER TRIGGER DeleteOrderOnReservationRejection
ON Reservations
FOR UPDATE
AS BEGIN
   SET NOCOUNT OFF
   DECLARE @reservationId INT
   DECLARE @status VARCHAR(MAX)


   SELECT TOP 1
       @reservationId = inserted.ReservationId,
       @status = inserted.Status
   FROM inserted

   PRINT 'Reservation [' + CAST(@reservationId as VARCHAR(max) )+
'] update'

   IF @status <> 'Rejected'
       RETURN;

   --- DELETE ALL ORDER DETAILS ---

   DELETE FROM OrderDetails WHERE OrderDetailId IN (
       SELECT OrderDetailId FROM OrderDetails
           INNER JOIN Orders
       ON OrderDetails.OrderId = Orders.OrderId
       WHERE Orders.ReservationId = @reservationId
   );

   --- DELETE ALL ORDERS ---

   DELETE FROM Orders WHERE OrderId IN (
       SELECT OrderId FROM Orders
       WHERE Orders.ReservationId = @reservationId
   );

   --- RESULT ---

   PRINT 'Deleted orders: ' + CAST(@@ROWCOUNT as VARCHAR(MAX))
END
```

Trigger ValidatePremiumMealOrdered

Trigger blokuje zamówienia, które zawierają produkty premium a rezerwacja zamówienia z tymi produktami została złożona za późno, także sprawdza czy rezerwacja jest na jeden z dni czwartek, piątek, sobota. (Musi być do poniedziałku poprzedzającego).

```
CREATE TRIGGER [dbo].[ValidatePremiumMealOrdered]
ON
[dbo].[OrderDetails]
after insert as

BEGIN
  -- zawsze będzie tylko jeden taki orderid, bo jest onInsert


  DECLARE @OrderToRemove int = (SELECT DISTINCT OD.Orderid FROM
OrderDetails AS OD
  INNER JOIN Menu AS MN ON MN.MenuMealId = OD.MenuMealId
  INNER JOIN Meals AS M ON MN.MealId = M.MealId
  INNER JOIN Orders AS O ON O.OrderId = OD.OrderId
  INNER JOIN Reservations AS R ON R.ReservationId =
O.ReservationId
  WHERE M.IsPremium = 1
  AND O.ReservationId IS NOT NULL
  AND (DATEPART(WEEKDAY, R.StartDate) NOT BETWEEN 4 AND 6
  OR O.OrderDate >= (SELECT DATEADD(week, DATEDIFF(week, 0,
DATEADD(DAY, -1,R.StartDate)), 0)) ))

  IF @OrderToRemove IS NULL
  BEGIN
     SET @OrderToRemove = (SELECT DISTINCT OD.Orderid FROM
OrderDetails AS OD
  INNER JOIN Menu AS MN ON MN.MenuMealId = OD.MenuMealId
  INNER JOIN Meals AS M ON MN.MealId = M.MealId
  INNER JOIN Orders AS O ON O.OrderId = OD.OrderId
  INNER JOIN TakeawayOrders AS T ON T.OrderId = OD.OrderId

  WHERE M.IsPremium = 1
  AND (DATEPART(WEEKDAY, T.TakeawayDate) NOT BETWEEN 4 AND 6
  OR O.OrderDate >= (SELECT DATEADD(week, DATEDIFF(week, 0,
DATEADD(DAY, -1,T.TakeawayDate)), 0)) ))



  DELETE FROM TakeawayOrders WHERE OrderId = @OrderToRemove

  DELETE FROM OrderDetails WHERE OrderId = @OrderToRemove
```

```
  DELETE
  FROM ORDERS
  WHERE OrderId = @OrderToRemove


  PRINT 'Cannot order seafood'
  END
  ELSE
  BEGIN

     DELETE
     FROM OrderDetails WHERE OrderId = @OrderToRemove

     DELETE
     FROM Orders WHERE OrderId = @OrderToRemove

     PRINT 'Cannot order seafood'

  END
END
go


ALTER TABLE [dbo].[OrderDetails] ENABLE TRIGGER
[ValidatePremiumMealOrdered]
GO
```

Indeksy

Addresses

Tabela bedzie dość często użytkowana (użytkowana w funkcjach, widokach) więc warto w nią ‘zainwestować’ więcej miejsca.

Address

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Address
ON Addresses(AddressId)
```

AddressToCity

Indeks przyśpieszający łączenie tablic.

```
CREATE INDEX AddressToCity
ON Addresses(CityId)
```

AddressStreet

```
CREATE NONCLUSTERED INDEX AddressStreet
ON Addresses(Street)
```

AddressNumber

```
CREATE NONCLUSTERED INDEX AddressNumber
ON Addresses(HouseNumber)
```

BigDiscounts

Tabela sama w sobie będzie tabelą często odczytywaną więc nie ma sensu używać dodawać indeksów non-clustered (oszczędność miejsca).

BigDiscount

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX BigDiscount
ON BigDiscounts(BigDiscountId)
```

Categories

Tabela używana w kilku widokach gdzie odczytywana jest nazwa kategorii. Nie jakoś bardzo często ale ze względu że będzie zawierała mało danych warto dołożyć więcej indeksów nie klastrowych.

Category

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Category
ON Categories(CategoryId)
```

CategoryName

```
CREATE NONCLUSTERED INDEX CategoryName
ON Categories(Name)
```

Cities

Tabela często używana do łączenia się z adresem przy odczytywaniu informacji do zamówienia lub o kliencie ogólnie.

City

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX City
ON Cities(CityId)
```

PostalCode

```
CREATE NONCLUSTERED INDEX PostalCode
ON Cities(PostalCode)
```

Name

```
CREATE NONCLUSTERED INDEX Name
ON Cities(Name)
```

CompanyCustomers

Tabela często używana i łączona dlatego posiada wiele indeksów.

CompanyCustomer

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX CompanyCustomer
ON CompanyCustomers(CustomerId)
```

CompanyNIP

```
CREATE UNIQUE NONCLUSTERED INDEX CompanyNIP
ON CompanyCustomers(NIP)
```

CompanyName

```
CREATE UNIQUE NONCLUSTERED INDEX CompanyName
ON CompanyCustomers(CompanyName)
```

Address

```
CREATE UNIQUE NONCLUSTERED INDEX Address
ON CompanyCustomers(AddressId)
```

Customers

Tabela używana w wielu widokach.

Customer

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Customer
ON Customers(CustomerId)
```

CustomersEmail

```
CREATE UNIQUE NONCLUSTERED INDEX CustomersEmails
ON Customers(Email)
```

CustomersPhoneNumber

```
CREATE UNIQUE NONCLUSTERED INDEX CustomersPhoneNumber
ON Customers(PhoneNumber)
```

IndividualCustomers

Customer

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Customer
ON IndividualCustomers(CustomerId)
```

FirstName

```
CREATE NONCLUSTERED INDEX FirstName
ON IndividualCustomers(FirstName)
```

LastName

```
CREATE NONCLUSTERED INDEX Name
ON IndividualCustomers(LastName)
```

IndividualCustomersAddresses

Założenia wiemy że wielu klientów nie poda adresu więc stworzenie dodatkowego indeksu nie wykorzysta tak dużo miejsca a może przyspieszyć niektóre widoki i funkcje.

CustomerAddress

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Customer
ON IndividualCustomersAddresses(CustomerId)
```

Address

```
CREATE NONCLUSTERED INDEX Address
ON IndividualCustomersAddresses(AddressId)
```

Discounts

Tabela sama w sobie będzie tabelą często odczytywaną więc nie ma sensu używać dodawać indeksów non-clustered (oszczędność miejsca).

Discount

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Discount
ON Discounts(DiscountId)
```

Employees

Tabela rzadko używana zawiera niewiele indeksów

Employee

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Employee
ON Discounts(DiscountId)
```

EmployeesName

```
CREATE NONCLUSTERED INDEX EmployeesNames
ON Employees(LastName,FirstName)
```

EmployeesNameReversed

```
CREATE NONCLUSTERED INDEX EmployeesNamesReversed
ON Employees(FirstName,LastName)
```

EmployeeDetails

Tabela łącząca 2 klucze głowne.

AssignedEmployee

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX AssignedEmployee
ON EmployeesDetails(EmployeeId, OrderId)
```

LoyaltyCards

Tabela bardzo często odczytywana zawiera kilka indeksów poprawiających jej działanie.

LoyaltyCard

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX LoyaltyCard
ON LoyaltyCards(LoyaltyCardId)
```

Customer

```
CREATE UNIQUE NONCLUSTERED INDEX Customer
ON LoyaltyCards(CustomerId)
```

Discount

```
CREATE NONCLUSTERED INDEX Discount
ON LoyaltyCards(DiscountId)
```

Meals

Tabela rzadko używana - liczba indeksów ograniczona aby oszczędzić miejsce.

Meal

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Meal
ON Meals(MealId)
```

MealName

```
CREATE UNIQUE NONCLUSTERED INDEX MealName
ON Meals(name)
```

Menu

Tabela bardzo często używana wiec zawiera kluczowe indeksy ponieważ wielkość samej tabeli bedzie stale rosnac a indeksy poprawia wydajność nie az tak bardzopowiekszajac potrzebne miejsce.

MealInMenu

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX MealInMenu
ON Menu(MenuMealId)
```

Meal

```
CREATE NONCLUSTERED INDEX Meal
ON Menu(MealId)
```

OneTimeDiscounts

Tabela bardzo często odczytywana zawiera kilka indeksów poprawiających jej działanie.

OneTimeDiscount

Domyślny indeks.

```
CREATE CLUSTERED UNIQUE INDEX OneTimeDiscount
ON OneTimeDiscounts(OneTimeDisocuntId)
```

Customer

```
CREATE NONCLUSTERED UNIQUE INDEX Customer
ON OneTimeDiscounts(CustomerId)
```

BigDiscount

```
CREATE NONCLUSTERED INDEX BigDiscount
ON OneTimeDiscounts(BigDiscountId)
```

OrderDetails

Tabela zawiera jeden indeks nieklastrowany aby ograniczyć miejsce użytkowane przez wpis których w bazie będzie dużo. Tabela często jest łącznikiem zamówień i Menu.

OrderDetail

Domyślny indeks.

```
CREATE CLUSTERED INDEX OrderDetail
ON OrderDetails(OrderId, MenuMealId)
```

OrderDetailsToMenu

```
CREATE NONCLUSTERED INDEX OrderDetailsToMenu
ON OrderDetails(OrderId, MenuMealId)
```

Orders

Tabela bardzo często używana więc zawiera dość dużo indeksów.

Order

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX OrderPK
ON Orders(OrderId)
```

OrderToReservation

```
CREATE NONCLUSTERED INDEX OrderToReservation
ON Orders(ReservationId)
```

OrderToCustomer

```
CREATE NONCLUSTERED INDEX OrderToCustomer
ON Orders(CustomerId)
```

ReservationDetails

Prosty łacznik zawiera tylko jeden indeks.

AssignedTable

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED  INDEX AssignedTable
ON ReservationDetails(ReservationId, TableId)
```

Reservations

Dość duża tabela zawiera tylko indeksy na najważniejsze pola.

ReservationsCustomers

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Reservation
ON Reservations(ReservationId)
```

ReservationsCustomers

```
CREATE NONCLUSTERED INDEX ReservationsCustomers
ON Reservations(CustomerId)
```

ReservationVariables

Prosta tabela ze stałymi nie potrzebny inny indeks niż na klucz główny tabeli.

Variables

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Variables
ON ReservationVariables(Id)
```

TakeawayOrders

Tabela w stosunku do innych nie będzie zajmować aż tyle miejsca więc dołożyliśmy tutaj dodatkowy indeks na id zamówienia.

TakeawayOrder

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX TakeawayOrder
ON TakeawayOrders(TakeawayOrderId)
```

TakeawayOrderToOrder

```
CREATE UNIQUE NONCLUSTERED INDEX TakeawayOrderToOrder
ON TakeawayOrders(OrderId)
```

Tables

Tabela nie jest wielka w stosunku do innych więc wszystkie jej pola są indeksami.

Table

Domyślny indeks.

```
CREATE UNIQUE CLUSTERED INDEX Table
ON Tables(size)
```

TablesSize

```
CREATE NONCLUSTERED INDEX TablesSizes
ON Tables(size)
```

Uprawnienia

Administrator

Osoba zarządzająca systemem. Brak ograniczeń.

```
CREATE ROLE admin
grant all privileges ON u_jkedra.dbo TO admin
```

Moderator

Osoba mogąca wprowadzać zmiany w bazie danych, ale która nie powinna mieć wpływu na strukturę bazy danych oraz nie posiada uprawnień do korzystania z funkcji oraz procedur.

```
CREATE ROLE moderator
GRANT SELECT, INSERT, UPDATE, DELETE ON DATABASE::u_jkedra TO
moderator
```

Manager

Osoba zatrudniona w restauracji na wyższym stanowisku. Ma dostęp odczytu do większej liczby tabel, prawo do ich modyfikacji oraz prawo do wykonywania wszystkich dostępnych funkcji oraz procedur.

```
CREATE ROLE manager
GRANT SELECT, INSERT, UPDATE, DELETE, EXECUTE ON
DATABASE::u_jkedra TO manager
```

Pracownik

Osoba zatrudniona w restauracji. Ma podstawowe prawa potrzebne do wykonywania najprostszych czynności.

```
CREATE ROLE worker

GRANT SELECT ON MenuView to worker
GRANT SELECT ON ClientsAwaitingPayments to worker
GRANT SELECT ON OrdersAwaitingPayment to worker
GRANT SELECT ON OrdersInfo to worker
GRANT SELECT ON DiscountInfo to worker
GRANT SELECT ON EmptyTables to worker
GRANT SELECT ON PendingReservations to worker

GRANT EXECUTE ON addEmployeeToOrder to worker
GRANT EXECUTE ON addTableToReservation to worker
```

```
GRANT EXECUTE ON AddAddress to worker
GRANT EXECUTE ON AddCity to worker
GRANT EXECUTE ON AddOrder to worker
GRANT EXECUTE ON AddOrderDetail to worker
GRANT EXECUTE ON ModifyOrderDetail to worker
GRANT EXECUTE ON AddTakeawayOrder to worker
GRANT EXECUTE ON changeAddress to worker
GRANT EXECUTE ON addEmployeeToOrder to worker
GRANT EXECUTE ON ModifyTable to worker
GRANT EXECUTE ON PayForOrder to worker


GRANT EXECUTE ON GetCustomersById to worker
GRANT EXECUTE ON getEmployees to worker
GRANT EXECUTE ON GetMealsSoldAtLeastXTimes to worker
GRANT EXECUTE ON GetMenuByDate to worker
GRANT EXECUTE ON GetOrderInformations to worker
GRANT EXECUTE ON GetXBestMeals to worker
GRANT EXECUTE ON IsMenuCorrect to worker
```

Przykładowe dane do testowania:

AddOrder

Zamówienie z rezerwacją

```
DECLARE @reservationId int
DECLARE @orders FullOrder

SET @reservationId = 38

INSERT INTO @orders (mealId, quantity)
VALUES
   (62, 19),
   (34, 12)

EXECUTE AddOrder @reservationId = @reservationId , @orders=@orders
```

Zamówienie na wynos

```
DECLARE @takeawayDate DATETIME
SET @takeawayDate = DATEADD(week, 1, GETDATE())

EXECUTE AddOrder @takeawayDate = @takeawayDate, @customerId = 0 ,
@orders=@orders
```

Zamówienie z rezerwacją z jednoczesną datą odbioru (error)

```
DECLARE @reservationId int = 38
DECLARE @orders FullOrder

INSERT INTO @orders (mealId, quantity)
VALUES
   (62, 19),
   (34, 12)

DECLARE @takeawayDate DATETIME
SET @takeawayDate = DATEADD(week, 1, GETDATE())

EXECUTE AddOrder @takeawayDate = @takeawayDate, @customerId = 0 ,
@orders=@orders, @reservationId = @reservationId
```

Zamówienie z rezerwacja z produktem spoza menu (error)

```
DECLARE @reservationId int
DECLARE @orders FullOrder
```

```
SET @reservationId = 38

INSERT INTO @orders (mealId, quantity)
VALUES
   (62, 19),
   (34, 12),
   (3300, 12)

EXECUTE AddOrder @reservationId = @reservationId , @orders=@orders
```

Zamówienie z rezerwacja z testem na quantity (error)

```
DECLARE @reservationId int
DECLARE @orders FullOrder

SET @reservationId = 38

INSERT INTO @orders (mealId, quantity)
VALUES
   (62, 19),
   (34, 12),
   (33, 1200000)

EXECUTE AddOrder @reservationId = @reservationId , @orders=@orders
```

RefreshMenu + isCorrect

Test RefreshMenu oraz funkcji isMenuCorrect

```
DECLARE @date DATETIME = DATEADD(day, 0, GETDATE())
PRINT 'Before Refresh: ' + Cast(dbo.IsMenuCorrectByDate (@date) as
VARCHAR(MAX) )

EXEC RefreshMenu @date

PRINT 'After Refresh: ' + Cast(dbo.IsMenuCorrectByDate (@date) as
VARCHAR(MAX) )
```

Trigger test

Test triggera na seefoods

```
USE [u_jkedra]
GO


DECLARE @products FullOrder
INSERT @products Values (1, 68)
DECLARE    @return_value int

EXEC   @return_value = [dbo].[AddIndividualReservation]
     @CustomerId = 100,
     @Products = @products,
     @Seats = 3,
     @StartDate = N'2023-01-17 00:00:00.000',
     @Invoice = 0,
     @isPaid = 0

SELECT 'Return Value' = @return_value

GO

select * from Orders where ReservationId = 50
```

AddIndividualReservation

Test dodawania rezerwacji indywidualnej - poprawna rezerwacja

```
USE [u_jkedra]
GO

DECLARE @list FullOrder
Insert @list values (1, 25)

DECLARE    @return_value int

EXEC   @return_value = [dbo].[AddIndividualReservation]
     @CustomerId = 100,
     @Products = @list,
     @Seats = 3,
     @StartDate = N'2023-01-18 00:00:00.000',
     @Invoice = 0,
     @isPaid = 0
```

```
SELECT 'Return Value' = @return_value

GO

select * from Reservations inner join orders on
orders.ReservationId = Reservations.ReservationId
```

RejectReservation

Test odrzucenia rezerwacji

```
USE [u_jkedra]
GO

DECLARE    @return_value int

EXEC   @return_value = [dbo].[RejectReservation]
     @ReservationId = 49

SELECT 'Return Value' = @return_value

GO


select * from Reservations left join Orders on
Orders.ReservationId = Reservations.ReservationId left join
OrderDetails on OrderDetails.OrderId = Orders.OrderId
```

AddCompanyReservation

Test rezerwacji firmowej

```
USE [u_jkedra]
GO

DECLARE    @return_value int

EXEC   @return_value = [dbo].[AddCompanyReservation]
     @CustomerId = 134,
     @Seats = 12,
     @StartDate = N'2023-01-18 00:00:00.000'
```

```
SELECT 'Return Value' = @return_value

GO

select * from Reservations
```

AddCompanyReservation

Test rezerwacji firmowej z listą gości

```
-- Add CompanyReservation (z listą gości)
USE [u_jkedra]
GO

DECLARE @guestList GuestList
Insert @guestList values (144), (145), (146)
DECLARE    @return_value int

EXEC   @return_value = [dbo].[AddCompanyReservation]
     @CustomerId = 134,
     @Seats = 3,
     @StartDate = N'2023-01-18 00:00:00.000',
     @Guests = @guestList

SELECT 'Return Value' = @return_value

GO

select * from Reservations inner join  CompanyCustomerList on
Reservations.ReservationId = CompanyCustomerList.ReservationIds
```

AcceptReservation

Test akceptacji rezerwacji

```
USE [u_jkedra]
GO

DECLARE @tables TableList
Insert @tables values (15)

DECLARE    @return_value int
```

```
EXEC   @return_value = [dbo].[AcceptReservation]
     @ReservationId = 48,
     @Tables = @tables

SELECT 'Return Value' = @return_value

GO

select * from Reservations inner join orders on
orders.ReservationId = Reservations.ReservationId inner join
ReservationDetails on ReservationDetails.ReservationId =
Reservations.ReservationId
```