Systemy Baz Danych

2023/2024 – projekt

Authors: *Urszula Stankiewicz, Michalina Hytrek, Łukasz Kwinta*

1. Opis systemu

Z tworzonej bazy danych skorzysta firma oferująca różnego rodzaju kursy i szkolenia:

- webinary - odbywają się na żywo na jednej z platform chmurowych, a ich nagrania są udostępniane klientom firmy. Nagrania nie są przechowywane w bazie - jedynie informacja o nich, którą usunąć może administrator.
- kursy - krótkie formy kształcenia, trwające zazwyczaj kilka dni, istnieją wyłącznie kursy płatne. Zaliczenie kursu wymaga zaliczenia min. 80% modułów.
- studia - kilkuletnie szkolenia odbywające się online i stacjonarnie, wymagają zaliczenia praktyk i zdania egzaminu końcowego

Każda z tych form kształcenia prowadzona jest przez konkretnego wykładowcę w konkretnym języku (najczęściej polskim). Czasami treść jest tłumaczona na żywo przez tłumacza, co też powinno zostać odnotowane w bazie danych.

Możemy wyróżnić następujących aktorów systemu:

- Klient - użytkownik chcący skorzystać z oferty firmy szkoleniowej
- Właściciel - osoba tworząca materiały video i treść kursów
- Administrator - zarządzanie bazą danych oraz jej ulepszanie Aktorzy mogą skorzystać z następujących funkcjonalności:

1.1 Klient

1.1.1. Webinary

- Korzystanie z nagrań bezpłatnych webinarów przez okres 30 dni od ich umieszczenia na stronie
- Użytkownicy posiadający konto: Po opłaceniu dostępu do webinarów płatnych, korzystanie z nagrań tych webinarów przez kolejne 30 dni od potwierdzenia opłaty
-

1.1.2. Kursy

- Kontrolowanie zaliczenia danego kursu (procent zaliczonych modułów >= 80 %) Sprawdzenie statusu swojej obecności na wybranych modułach
- Dostęp do listy kursów na które użytkownik jest zapisany i dostęp do statusu płatności przy każdym kursie (nieopłacone/ zaliczka/ opłacone w całości)
- Sprawdzenie dostępności wolnych miejsc na kursy hybrydowe i stacjonarne
- Dostęp do dodatkowych informacji o kursach takich jak: język kursu, obecność tłumacza, sposobie organizacji kursu (stacjonarnie/ o-line synchronicznie/ online asynchronicznie/ hybrydowo), dacie rozpoczęcia kursu czy sali zajęciowej (informacja dostępna po uiszczeniu wszelkich opłat) Dostęp do nagranych modułów (moduły online), po opłaceniu dostępu

1.1.3. Studia

- Sprawdzenie swojej obecności na zajęciach
- Możliwość zapisania się na odrabianie zajęć w kursie lub zajęciach innego kursu o podobnej tematyce
- Sprawdzenie wyników z egzaminów
- Sprawdzenie informacji o tym, czy odbyło się praktyki (14 dni - 2 razy w ciągu roku) i frekwencji na nich
- Możliwość zapisania się na pojedyncze zajęcia
- Wyświetlenie sylabusu studiów

1.1.4. Koszyk

- dodawanie produktów do koszyka (kursy, webinary, studia)

1.2. Sekretarz

- Wyświetlanie następujących raportów:
  - lista osób, które skorzystały z oferty firmy, ale za to nie zapłaciły
  - lista osób zapisanych na przyszłe wydarzenia z informacją, czy wydarzenia te odbywają się stacjonarnie, czy online
  - raport dotyczący frekwencji na wydarzeniach przeszłych - liczba osób które brały udział w każdym kursie/webinarze/studium i były obecne
  - lista osób, które są zapisane na kolidujące ze sobą wydarzenia
  - lista wyników egzaminów dla użytkowników
  - lista obecności na zajęciach dla danego użytkownika
  - lista odbytych praktyk
  - Dodanie nowego klienta
- Wyświetlanie spisu wszystkich zajęć i wszystkich spotkań z datami

1.3. Manager

Funkcje jakie ma sekretarz + dodatkowo:

- Wyświetlanie następujących raportów:
    - finansowe - zestawienie przychodów dla każdego kursu/studium/webinaru - przesyłana jest informacja o tym do właściciela
    - lista osób zapisanych na każde szkolenie zawierająca imię, nazwisko, informacja, czy klient był obecny
- Wyświetlanie spisu wszystkich zajęć i wszystkich spotkań z datami oraz możliwość ich zmiany (studia)
- Określenie limitu miejsc na kursy hybrydowe/stacjonarne oraz studia
- Możliwość generowania listy klientów którzy są uprawnieni do otrzymania dyplomów (ukończyli kurs/studia)

1.4. Nauczyciel

- Dodawanie nagrań szkoleń
- Dostęp do prowadzonych przez siebie nagrań i list obecności z prowadzonych przez siebie zajęć

1.5. Właściciel

Funkcje managera i sekretarza + dodatkowo:

- Zezwalanie na odroczenie płatności za szkolenia

1.6. Funkcje systemu

1.6.1. Webinary

- kontrola dostępu klientów do webinarów
    - webinary bezpłatne - dostęp przez 30 dni od umieszczenia nagrania na platformie
    - webinary płatne - dostęp przez 30 dni od uiszczenia opłaty
    - uniemożliwienie korzystania z płatnych webinarów użytkownikom niezalogowanym i tym, którzy nie uiścili opłaty

1.6.2. Kursy

- weryfikacja zaliczenia danych modułów wchodzących w skład kursu
- kontrola dostępu klientów do kursów:
    - kursy on-line synchronicznie (zasady jak przy webinarach)
    - kursy online asynchronicznie (dostęp po dodaniu materiałów przez właściciela i po uiszczeniu opłat przez klienta)
    - uniemożliwienie dostępu do kursów on-line użytkownikom którzy nie wpłacili całości kwoty 3 dni przed rozpoczęciem kursu

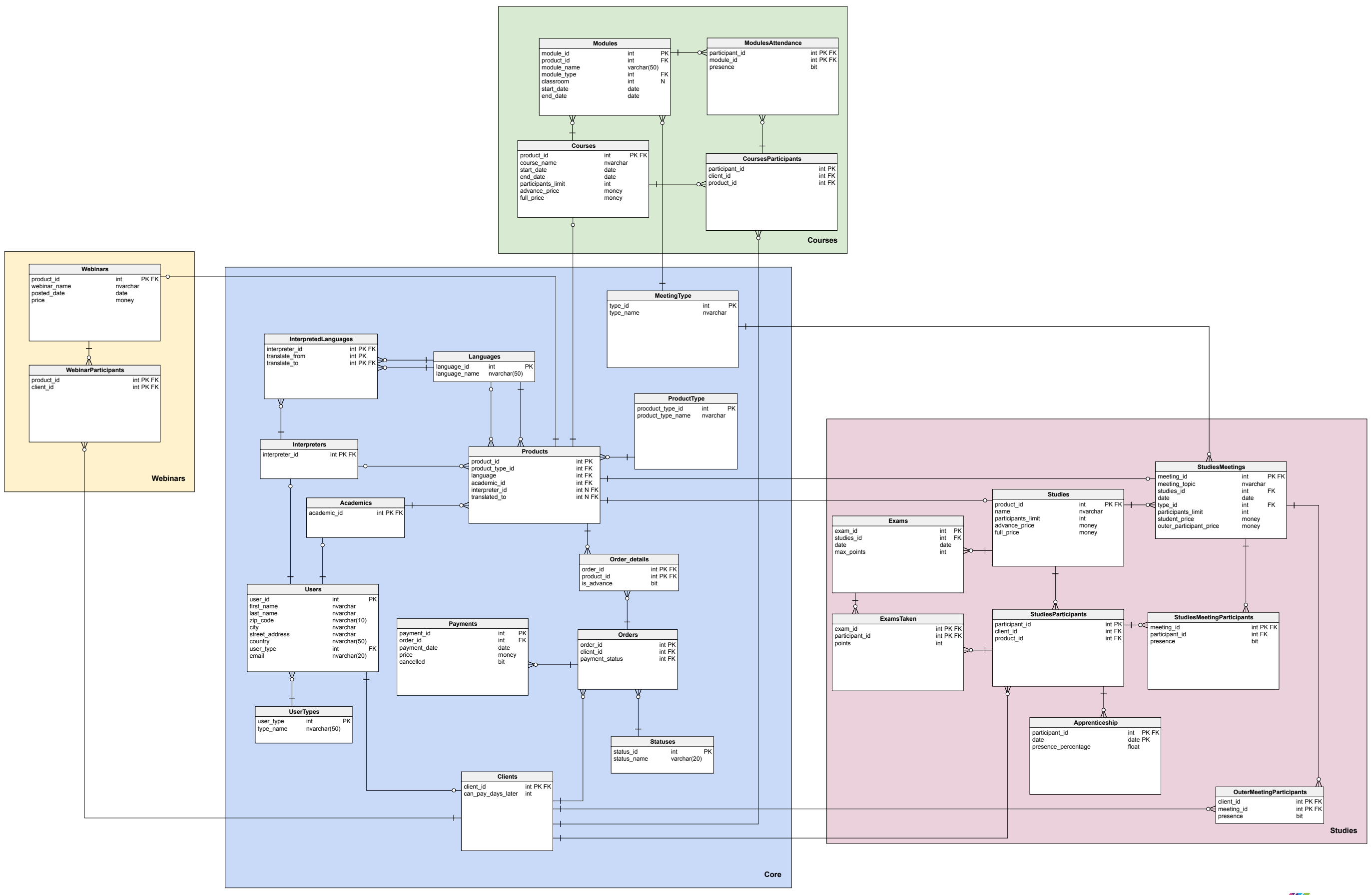
1.6.3. Studia

- kontrola dostępu klientów do studiów
    - spotkania on-line
    - spotkania stacjonarnie
    - spotkania hybrydowe
    - możliwość wykupienia dostępu płatnego do jednego spotkania
    - limit miejsc ogólny (nie może być większy niż najmniejszy spośród limitów wszystkich spotkań)
- przechowywanie informacji o sylabusie (przechowywanie listy zajęć na danym studium i listy różnych studiów jeszcze przed danym rokiem)
- przechowywanie informacji o spisie wszystkich zajęć i wszystkich spotkań z datami
    - limit miejsc na spotkanie
- kontrola, czy studenci zaliczyli praktyki trwające 14 dni - 2 razy w ciągu roku
- kontrola obecności klientów na spotkaniach i praktykach
    - aby zaliczyć studium:
        - 80% obecności na spotkaniach
        - 100% obecności na praktykach
- kontrola, czy studenci uiścili opłatę wpisową oraz za każde spotkanie najpóźniej 3 dni przed zjazdem
- przyznawanie statusu zaliczenia i ew. wysłania dyplomu Pocztą polska na status korespondencyjny (na podstawie zaliczenia praktyk i egzaminu końcowego oraz obecności)

1.6.4. Koszyk

- po kliknięciu przez klienta "Zakończ i zapłać", wygenerowanie linku do płatności.
- po zakończeniu transakcji przesłanie informacji zwrotnej o pomyślnym zakończeniu płatności lub błędzie.

2. Schemat bazy danych



3. Implementacje tabel

3.1 Core

Główna część systemu

Users

Zawiera wszystkich użytkowników systemu oraz ich dane - imię, nazwisko, dane adresowe oraz typ użytkownika (klucz obcy do tabeli User_types), a także informację o tym, ile dni opóźnienia w płatności jest dozwolone danemu użytkownikowi.

```
create table Users
(
    user_id        int identity
        constraint user_id
            primary key,
    first_name     nvarchar(50)           not null,
    last_name      nvarchar(50)           not null,
    zip_code       nvarchar(10)           not null,
    city           nvarchar(50)           not null,
    street_address nvarchar(50)           not null,
    country        nvarchar(50)           not null,
    user_type      int
        constraint df_user_type default 1 not null
        constraint User_types_Users
            references User_types
            on update cascade on delete cascade,
    email          nvarchar(50)           not null
        constraint email_unique
            unique
        constraint ValidEmail
            check ([Email] like '%_@__%.__%')
)
go

create index Users_last_name_index
    on Users (last_name)
go
```

```
create index Users_zip_code_index
    on Users (zip_code)
go

create index Users_country_index
    on Users (country)
go
```

Academics

Zawiera id wszystkich użytkowników, którzy są nauczycielami - zdecydowaliśmy się na dodanie tabel Academics, Interpreters i Clients, by rozdzielić logikę wykonywaną dla poszczególnych typów użytkownika.

```
create table Academics
(
    academic_id int not null
        constraint Academics_pk
            primary key
        constraint FK_Academics_Users
            references Users
            on update cascade on delete cascade
)
go
```

Interpreters

Zawiera id wszystkich tłumaczy

```
create table Interpreters
(
    interpreter_id int not null
        constraint Interpreters_pk
            primary key
        constraint Interpreters_Users
            references Users
            on update cascade on delete cascade
)
go
```

Clients

Zawiera id wszystkich klientów

```
create table Clients
(
    client_id          int                         not null
        constraint client_id
            primary key
        constraint Clients_Users
            references Users,
    can_pay_days_later int
        constraint df_Can_pay_days_later default 0 not null
)
go
```

User_types

Zawiera listę wszystkich typów użytkowników występujących w systemie

```
create table User_types
(
    user_type int identity
        constraint User_types_pk
            primary key,
    type_name nvarchar(50) not null
)
go
```

Interpreted_languages

Każdemu tłumaczowi przyporządkowuje informację o tym, z jakiego języka na jaki tłumaczy (są to FK do tabeli languages)

```
create table Interpreted_languages
(
    interpreter_id int not null
        constraint Interpreted_languages_Interpreters
            references Interpreters
            on update cascade on delete cascade,
    translate_from int not null
        constraint FK_Interpreted_languages_Languages
            references Languages
        constraint FK_Interpreted_languages_Languages2
            references Languages,
    translate_to   int not null
        constraint FK_Interpreted_languages_Languages1
            references Languages,
    constraint Interpreted_languages_pk
        primary key (interpreter_id, translate_from, translate_to)
)
go
```

Languages

Lista wszystkich języków, w jakich prowadzone są szkolenia, bądź na jakie są one tłumaczone

```
create table Languages
(
    language_id   int identity
        constraint PK_Languages
            primary key,
    language_name nvarchar(50) not null
        constraint language_name_unique
            unique
)
go
```

Products

Zawiera wszystkie produkty, informację o ich typie (odwołanie do tabeli ProductType), języku w jakim jest prowadzone dane szkolenie, wykładowcy, który je prowadzi oraz o tłumaczu i języku, na który tłumaczone jest szkolenie

```
create table Products
(
    product_id      int identity
        constraint Products_pk
            primary key,
    product_type_id int not null
        constraint Products_ProductType
            references ProductType
            on update cascade on delete cascade,
    language        int not null
        constraint FK_Products_Languages
            references Languages,
    academic_id     int not null
        constraint FK_Products_Academics
            references Academics
            on update cascade on delete cascade,
    interpreter_id  int
        constraint FK_Products_Interpreters1
            references Interpreters,
    translated_to   int
        constraint FK_Products_Languages1
            references Languages
)
```

```
go

create index Products_product_type_id_index
    on Products (product_type_id)
go

create index Products_language_index
    on Products (language)
go
```

ProductType

Zawiera wszystkie typy produktów (webinary, spotkania, kursy, studia)

```
create table ProductType
(
    product_type_id   int identity
        constraint ProductType_pk
            primary key,
    product_type_name nvarchar(50) not null
)
go
```

Payments

Spis wszystkich płatności (numer zamówienia, data płatności, wpłacona kwota)

```
create table Payments
(
    payment_id   int identity
        constraint Payments_pk
            primary key,
    order_id     int                              not null
        constraint Orders_Payments
            references Orders,
    payment_date date                             not null
        constraint payment_date_check
            check ([payment_date] >= '1990-01-01' AND [payment_date] <= getdate()),
    price        money                            not null,
    cancelled    bit
        constraint DF_Payments_cancelled default 0 not null
)
go

create index Payments_order_id_index
    on Payments (order_id)
go

create index Payments_payment_date_index
    on Payments (payment_date)
go
```

MeetingType

Rodzaje spotkań (online, hybrydowe, stacjonarne)

```
create table MeetingType
(
    type_id   int identity
        constraint type_id
            primary key,
    type_name nvarchar(50) not null
)
go
```

Orders

Lista wszystkich zamówień (numer klienta, status płatności)

```
create table Orders
(
    order_id       int identity
        constraint Orders_pk
            primary key,
    client_id      int                         not null
        constraint Orders_Clients
            references Clients,
    payment_status int
        constraint df_payment_status default 2 not null
        constraint Statuses_Orders
            references Statuses
)
go
```

OrdersDetails

Lista wszystkich zamówień (numer klienta, status płatności)

```
create table Order_details
(
    order_id   int                                       not null
        constraint Order_products_Orders
            references Orders,
    product_id int                                       not null
        constraint Order_products_Products
            references Products,
    is_advance bit
        constraint DF_Order_details_is_advance default 0 not null,
    constraint Order_details_pk
        primary key (order_id, product_id)
)
go
```

Statuses

Rodzaje statusów zamówień (nieopłacone, opłacone, częsciowo opłacone (z jakiegos produktu tylko zaliczka), anulowane)

```
create table Statuses
(
    status_id   int identity
        constraint Statuses_pk
            primary key,
    status_name varchar(20) not null
)
go
```

3.2. Webinars

Webinars

Lista wszystkich webinarów wraz z ich nazwami, datą publikacji i ceną

```
create table Webinars
(
    product_id   int                         not null
        constraint product_id_webinars
            primary key
        constraint Webinars_Products
            references Products
            on update cascade on delete cascade,
    webinar_name nvarchar(50)                not null,
    posted_date  date                        not null
        constraint check_posted_date
            check ([posted_date] >= '1990-01-01' AND [posted_date] <= getdate()),
    price        money
        constraint def_price default 0.00 not null
)
```

```
go

create index Webinars_webinar_name_index
    on Webinars (webinar_name)
go

create index Webinars_posted_date_index
    on Webinars (posted_date)
go
```

WebinarParticipants

Lista uczestników poszczególnych webinarów

```
create table WebinarParticipants
(
    product_id int not null
        constraint WebinarParticipants_Webinars
            references Webinars
            on update cascade on delete cascade,
    client_id  int not null
        constraint FK_WebinarParticipants_Clients
            references Clients
            on update cascade on delete cascade,
    constraint WebinarParticipants_pk
        primary key (client_id, product_id)
)
go
```

3.3. Courses

Courses

Lista kursów wraz z ich nazwami, datami początku i końca kursu, limitem uczestników, ceną zaliczki oraz pełną ceną

```
create table Courses
(
    product_id         int                        not null
        constraint product_id
            primary key
        constraint FK_Courses_Products
            references Products
            on update cascade on delete cascade,
    course_name        nvarchar(50)               not null,
    start_date         date                       not null,
    end_date           date                       not null,
    participants_limit int                        not null
        constraint participants_limit
            check ([participants_limit] >= 0),
    advance_price      money
    full_price         money
    constraint ch_advance_price
        check ([advance_price] < [full_price] AND [advance_price] >= 0),
    constraint ch_end_date
        check ([end_date] >= [start_date])
)
go

create unique index Courses_course_name_uindex
    on Courses (course_name)
go

create unique index Courses_start_date_end_date_uindex
    on Courses (start_date, end_date)
go
```

CoursesParticipants

Lista uczestników poszczególnych kursów

```sql
create table CoursesParticipants
(
    participant_id int identity
        constraint CoursesParticipants_pk
            primary key,
    client_id      int not null
        constraint CursesParticipants_Clients
            references Clients
            on update cascade on delete cascade,
    product_id     int not null
        constraint CoursesParticipants_Courses
            references Courses
)
go
```

Modules

Lista modułów kursów z nazwami, typem modułu (odwołanie do tabeli MeetingType), numerem sali oraz datą rozpoczęcia i zakończenia modułu

```sql
create table Modules
(
    module_id   int identity
        constraint Modules_pk
            primary key,
    product_id  int         not null
        constraint Courses_Modules
            references Courses
            on update cascade on delete cascade,
    module_name varchar(50) not null,
    module_type int         not null
        constraint Modules_MeetingType
            references MeetingType
            on update cascade on delete cascade,
    classroom   int,
    start_date  date        not null,
    end_date    date        not null,
    constraint ch_end_date_courses
        check ([end_date] >= [start_date])
)
go

create unique index Uniq_Modules
    on Modules (module_name)
go

create index Modules_product_id_index
    on Modules (product_id)
go

create index Modules_start_date_index
    on Modules (start_date)
go

create index Modules_classroom_index
    on Modules (classroom)
go
```

ModulesAttendance

Zawiera listę obecności uczestników kursów na poszczególnych modułach

```sql
create table ModulesAttendance
(
    participant_id int                                     not null
        constraint FK_ModulesAttendance_CoursesParticipants
            references CoursesParticipants
            on update cascade on delete cascade,
    module_id      int                                     not null
        constraint ModulesAttendance_Modules
            references Modules
            on update cascade on delete cascade,
    presence       bit
        constraint DF_ModulesAttendance_presence default 0 not null,
    constraint PK_ModulesAttendance
        primary key (participant_id, module_id)
)
go
```

3.4. Studies

Studies

Zawiera listę produktów typu "studia", nazwę studiów, limit uczestników oraz wysokość wpisowego

```
create table Studies
(
    product_id         int                               not null
        constraint studies_id
            primary key
        constraint Studies_Products
            references Products
            on update cascade on delete cascade,
    name               nvarchar(50)                      not null
        constraint check_name
            check (len([name]) > 0),
    participants_limit int default 100                   not null
        constraint check_praticipant_limit
            check ([participants_limit] > 0),
    full_price         money
        constraint check_full_price
            check ([full_price] >= 0),
    advance_price      money
    constraint check_advance_price
        check ([advance_price] <= [Studies].[full_price] AND [advance_price] >= 0)
)
go

create index Studies_name_index
    on Studies (name)
go
```

StudiesParticipants

Zawiera uczestników poszczególnych studiów

```
create table StudiesParticipants
(
    participant_id int identity
        constraint participant_id_studies_participants
            primary key,
    client_id      int not null
        constraint StudiesParticipants_Clients
            references Clients
            on update cascade on delete cascade,
    product_id     int not null
        constraint StudiesParticipants_Studies
            references Studies
)
go

create index StudiesParticipants_client_id_index
    on StudiesParticipants (client_id)
go

create index StudiesParticipants_product_id_index
    on StudiesParticipants (product_id)
go
```

Exams

Zawiera przypisane studiom egzaminy, datę odbycia się egzaminów oraz maksymalne możliwe do zdobycia punkty

```
create table Exams
(
    exam_id    int identity
        constraint PK_Exams
            primary key,
    studies_id int                    not null
        constraint Exams_Studies
            references Studies
            on update cascade on delete cascade,
    date       date default getdate() not null,
    max_points int  default 100       not null
        constraint check_max_points
            check ([max_points] > 0)
)
go

create index Exams_studies_id_index
    on Exams (studies_id)
go

create index Exams_date_index
    on Exams (date)
go
```

ExamsTaken

Zawiera dane odnośnie wyników egzaminów w których uczestnik studiów wziął udział

```
create table ExamsTaken
(
    exam_id        int            not null
        constraint ExamsTaken_Exams
            references Exams
            on update cascade
        constraint check_date
            check ([dbo].[checkExamDate]([exam_id]) <= getdate()),
    participant_id int            not null
        constraint ExamsTaken_StudiesParticipants
            references StudiesParticipants,
    points                       not null,
    constraint ExamsTaken_pk
        primary key (participant_id, exam_id),
    constraint check_points
        check ([points] >= 0 AND [points] <= [dbo].[checkExamMaxPoints]([exam_id]))
)
go
```

Apprenticeship

Zawiera uczestników, którzy odbyli praktyki w określonym terminie

```
create table Apprenticeship
(
    participant_id      int               not null
        constraint Apprenticeship_StudiesParticipants
            references StudiesParticipants
            on update cascade on delete cascade,
    date                date              not null,
    presence_percentage float default 100 not null
        constraint check_presence_percentage
            check ([presence_percentage] >= 0 AND [presence_percentage] <= 100)
)
go

create unique clustered index Apprenticeship_participant_id_date_uindex
    on Apprenticeship (participant_id, date)
go
```

StudiesMeetingParticipants

Zawiera listę obecnych studentów na danych spotkaniach

```
create table StudiesMeetingParticipants
(
    meeting_id     int not null
        constraint FK_MeetingParticipants_StudiesMeetings
            references StudiesMeetings
            on update cascade on delete cascade,
    participant_id int not null
        constraint MeetingParticipants_StudiesParticipants
            references StudiesParticipants
            on update cascade on delete cascade,
    presence       bit default 0,
    constraint meeting_id
        primary key (meeting_id, participant_id)
)
go
```

StudiesMeetings

Lista spotkań poszczególnych studiów, data spotkania, typ spotkania (FK do MeetingTypes), limit uczestników spotkania, cena dla studentów, cena dla uczestników, którzy nie są studentami

```
create table StudiesMeetings
(
    meeting_id                int                              not null
        constraint StudiesMeetings_pk
            primary key
        constraint StudiesMeetings_Products
            references Products
            on update cascade on delete cascade,
    studies_id                int                              not null
        constraint StudiesMeetings_Studies
            references Studies,
    date                      date                             not null,
    type_id                   int default 1                    not null
        constraint StudiesMeetings_MeetingType
            references MeetingType
            on update cascade on delete cascade,
    participants_limit                                        not null,
    student_price             money
        constraint check_student_price
            check ([student_price] >= 0),
    outer_participant_price money                              not null
        constraint check_outer_participant_price
            check ([outer_participant_price] >= 0),
    meeting_topic             nvarchar(50)                     not null
        constraint check_meeting_topic_length
            check (len([meeting_topic]) > 0),
    constraint check_participants_limit
        check ([dbo].[checkParicipantsLimit]([studies_id]) <= [StudiesMeetings].[participants_limit])
)
go

create index StudiesMeetings_studies_id_index
    on StudiesMeetings (studies_id)
go

create index StudiesMeetings_date_index
    on StudiesMeetings (date)
go
```

OuterMeetingsParticipants

Tabela zawierająca uczestników spotkań na studiach nie będących uczestnikami studiów

```
create table OuterMeetingParticipants
(
    client_id  int                                    not null
        constraint FK_OuterMeetingParticipants_Clients
            references Clients,
    meeting_id int                                     not null
        constraint FK_OuterMeetingParticipants_StudiesMeetings
            references StudiesMeetings,
    presence   bit
        constraint df_outer_meeting_presence default 0 not null,
    constraint PK_OuterMeetingParticipants
        primary key (client_id, meeting_id)
)
go
```

Widoki

Dla Sekretarza

BorrowersList

Lista klientów którzy skorzystali z oferty i za nią nie zapłacili (client_id, order_id)

```
CREATE VIEW [dbo].[BorrowersList] AS
    Select client_id, order_id
    From Orders as o
    Where order_id in ( Select order_id
                    From Order_details as od
                        inner join
                        (Select product_id as p_id, posted_date from Webinars where posted_date <= GETDATE()
                        UNION Select product_id as p_id, start_date from Courses where start_date <= GETDATE()
                        UNION Select studies_id as p_id, min(date) from StudiesMeetings group by studies_id
having (MIN(date)) <= GETDATE( ))
                        as p
                        on p.p_id=od.product_id)
                        and not( payment_status = 1)

go
```

PastEvents

Raport dotyczący frekwencji na danym wydarzeniu (moduł, spotkanie ze studiów) wraz z podstawowymi informacjami

```
CREATE VIEW [dbo].[PastEventsAttendance]
AS
SELECT p.product_id, pt.product_type_name as category, s.name as product_name, sm.meeting_id as id, sm.date as
date, mt.type_name as type, COUNT(mp.client_id) as attendance
FROM StudiesMeetings as sm
    inner join (SELECT participant_id as client_id, meeting_id
                FROM StudiesMeetingParticipants
                WHERE presence=1
                UNION
                SELECT client_id, meeting_id
                FROM OuterMeetingParticipants
                WHERE presence = 1) as mp
                on mp.client_id=sm.meeting_id
    inner join Studies as s on s.product_id=sm.studies_id and sm.date <= GETDATE()
    inner join Products as p on p.product_id=s.product_id
    join MeetingType as mt on mt.type_id=sm.type_id
    join ProductType as pt on pt.product_type_id=p.product_type_id
GROUP BY p.product_id, pt.product_type_name, s.name, sm.meeting_id, sm.date, mt.type_name
UNION
SELECT p.product_id, pt.product_type_name as category, c.course_name as product_name, m.module_id as id,
m.start_date as date, mt.type_name  as type, COUNT(ma.presence) as attendance
FROM Modules as m
    inner join ModulesAttendance as ma on m.module_id=ma.module_id and ma.presence=1
    inner join Courses as c on c.product_id=m.product_id and m.end_date <= GETDATE()
    inner join Products as p on p.product_id=c.product_id
    join MeetingType as mt on mt.type_id=m.module_type
    join ProductType as pt on pt.product_type_id=p.product_type_id
GROUP BY p.product_id, pt.product_type_name, c.course_name, m.module_id, m.start_date, mt.type_name
go
```

EventsThisMonth

Spis webinarów, modułów oraz spotkań ze studiów, które odbywają się w aktualnym miesiącu

```
CREATE VIEW [dbo].[EventsThisMonth]
AS
SELECT p.product_id, pt.product_type_name as category, s.name as product_name, sm.meeting_id as id, sm.date as
date, mt.type_name  as type
FROM StudiesMeetings as sm
    inner join Studies as s on s.product_id=sm.studies_id and YEAR(sm.date) = YEAR(GETDATE()) and
MONTH(sm.date) = MONTH(GETDATE())
    inner join Products as p on p.product_id=s.product_id
    join MeetingType as mt on mt.type_id = sm.type_id
    join ProductType as pt on pt.product_type_id=p.product_type_id
UNION
SELECT p.product_id, pt.product_type_name as category, w.webinar_name as product_name, w.product_id,
w.posted_date as date, 'on-line' as type
FROM Webinars as w
    inner join Products as p on p.product_id=w.product_id and YEAR(w.posted_date) = YEAR(GETDATE()) and
MONTH(w.posted_date) = MONTH(GETDATE())
    join ProductType as pt on pt.product_type_id=p.product_type_id
UNION
SELECT p.product_id, pt.product_type_name as category, c.course_name as product_name, m.module_id as id,
m.start_date as date, mt.type_name as type
FROM Modules as m
    inner join Courses as c on c.product_id=m.product_id and YEAR(m.start_date) = YEAR(GETDATE()) and
MONTH(m.start_date) = MONTH(GETDATE())
    inner join Products as p on p.product_id=c.product_id
    join MeetingType as mt on mt.type_id = m.module_type
    join ProductType as pt on pt.product_type_id=p.product_type_id
go
```

Exams Stats

Lista egzaminów wraz z srednia ilościa punktów uzyskanych przez studentów

```
CREATE VIEW ExamsStats
AS
SELECT e.studies_id as studies, e.exam_id as exam, e.max_points as max_points, AVG(et.points) as average_points
FROM Exams as e
    inner join ExamsTaken as et on et.exam_id=e.exam_id
GROUP BY e.studies_id, e.exam_id, e.max_points
go
```

StudentsApprenticeships

Lista studentów wraz z iloscią odbytych praktyk

```
CREATE VIEW StudentsApprenticeship
AS
SELECT a.participant_id, COUNT(a.date) as apprenticeships_taken
FROM Apprenticeship as a
GROUP BY a.participant_id
go
```

Bilocations

Lista osób zapisanych na kilka wydarzeń odbywajacych sie w tym samym czasie (client_id, date, num_of_events

```
CREATE VIEW [dbo].[Bilocations] As
    Select c.client_id, p.date, COUNT(p.date) as eventsNumber
    From Clients as c
    inner join Orders as o on c.can_pay_days_later=o.client_id
    inner join Order_details as od on od.order_id=o.order_id
    inner join( Select m.module_id as p_id, start_date as date from Modules as m where  not m.module_type = 1
            UNION
            Select sm.meeting_id as p_id, sm.date as date from StudiesMeetings as sm where not sm.type_id = 1
            UNION
            Select w.product_id as p_id, w.posted_date as date from Webinars as w
            UNION
            Select sm.student_price as p_id, sm.date as date from StudiesMeetings as sm where not sm.type_id =
1
            ) as p
            on p.p_id = od.product_id
    where p.date >= GETDATE()
    group by c.client_id, p.date
```

Dla Managera

Financial Report

Przedstawia podsumowanie finansowe

```
CREATE VIEW [dbo].[FinancialReport] AS
SELECT Products.product_id, dbo.getProductName(Products.product_id) AS product_name, product_type_name,
SUM(price) AS total_income
FROM Payments
         INNER JOIN Orders ON Payments.order_id = Orders.order_id
         INNER JOIN Order_details ON Orders.order_id = Order_details.order_id
         INNER JOIN Products ON Order_details.product_id = Products.product_id
         INNER JOIN ProductType ON Products.product_type_id = ProductType.product_type_id
GROUP BY Products.product_id, Products.product_id, product_type_name
go
```

GraduationCandidates

Przedstawia listę osób które zaliczyły studia lub kurs - są kandydatami do otrzymania certyfikatu

```
CREATE VIEW [dbo].[GraduationCandidates] AS
    SELECT Clients.client_id, first_name, last_name, dbo.getProductName(product_id) AS product_name
    FROM StudiesParticipants
        INNER JOIN Clients ON StudiesParticipants.client_id = Clients.client_id
        INNER JOIN Users ON Clients.client_id = Users.user_id
    WHERE dbo.studiesPass(participant_id) = 1
    UNION
    SELECT Clients.client_id, first_name, last_name, dbo.getProductName(product_id) AS product_name
    FROM CoursesParticipants
        INNER JOIN Clients ON CoursesParticipants.client_id = Clients.client_id
        INNER JOIN Users ON Clients.client_id = Users.user_id
    WHERE dbo.coursePass(participant_id) = 1
go
```

All Meetings

Wyświetla daty wszystkich spotkań

```
CREATE VIEW [dbo].[AllMeetings] AS
    SELECT product_id,'Module' AS type, module_name AS title, start_date AS date
    FROM Modules
    UNION
    SELECT studies_id,'Studies Meeting' AS type, meeting_topic AS title, date AS date
    FROM StudiesMeetings
    UNION
    SELECT product_id, 'Webinar' AS type, webinar_name, posted_date AS date
    FROM Webinars
go
```

Procedury

AddUser

Dodaje użytkownika o podanych danych (imię, nazwisko, adres,email, typ użytkownika)

```
CREATE PROCEDURE [dbo].[uspAddUser]
    @first_name nvarchar(50),
    @last_name nvarchar(50),
    @zip_code nvarchar(10),
    @city nvarchar(50),
    @street_address nvarchar(50),
    @country nvarchar(50),
    @email nvarchar(50),
    @type_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM User_types
            where @type_id=user_type
        )
        BEGIN
            ;
            THROW 52000, N'Taki rodzaj użytkownika nie istnieje',1
        END

        DECLARE @user_type_name nvarchar(50)
        SELECT @user_type_name = type_name
        FROM User_types
        WHERE  @type_id=user_type

        BEGIN TRANSACTION
            INSERT INTO Users (first_name,last_name,zip_code,city,street_address,country,user_type,email)

                    values(@first_name,@last_name,@zip_code,@city,@street_address,@country,@type_id,@email)

            DECLARE @user_id INT;
            SET  @user_id= SCOPE_IDENTITY();

            IF @user_type_name='client'
            Begin
                insert into clients (client_id)
                values(@user_id)
            end
            else IF @user_type_name='academic'
            Begin
                insert into academics (academic_id)
                values(@user_id)
            end
            else IF @user_type_name='interpreter'
            Begin
                insert into interpreters (interpreter_id)
                values(@user_id)
            end
        COMMIT TRANSACTION

    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd dodawania uzytkownika: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddWebinar

Dodaje webinar o podanej nazwie, id nauczyciela, nazwie języka oraz opcjonalnie danych o tłumaczu i języku, na który jest tłumaczone dane szkolenie do tabeli webinars oraz products

```
CREATE PROCEDURE [dbo].[uspAddWebinar]
    @language_id int,
    @academic_id int,
    @interpreter_id int=null,
    @translate_to_id int=null,
    @webinar_name nvarchar(50)
AS
BEGIN
```

```
    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Academics
            WHERE academic_id=@academic_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego nauczyciela!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @language_id=language_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @translate_to_id=language_id
        ) AND @translate_to_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Interpreters
            WHERE interpreter_id=@interpreter_id
        ) AND @interpreter_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego tłumacza!',1
        END

        DECLARE @type_id INT
        SELECT @type_id = product_type_id
        FROM ProductType
        WHERE 'webinar' = product_type_name

        BEGIN TRANSACTION
            INSERT INTO Products (product_type_id,language,academic_id,interpreter_id,translated_to)

                    values(@type_id,@language_id,@academic_id,@interpreter_id,@translate_to_id)

            DECLARE @product_id INT;
            SET  @product_id= SCOPE_IDENTITY();

            INSERT INTO Webinars(product_id,webinar_name, posted_date)
            Values (@product_id,@webinar_name, GETDATE());
        COMMIT TRANSACTION


    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd dodania webinaru: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

SetWebinarPrice

Zmienia cenę webinaru o podanej nazwie

```
CREATE PROCEDURE [dbo].[uspSetWebinarPrice]
    @webinar_id int,
    @price money
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM webinars
            where @webinar_id=product_id
        )
        BEGIN
            ;
            THROW 52000, N'Webinar o tej nazwie nie istnieje',1
        END

        UPDATE webinars
        SET price=@price
        where product_id=@webinar_id
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany ceny webinaru: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddCourse

Dodaje kurs o podanej nazwie, id nauczyciela, nazwie języka oraz opcjonalnie danych o tłumaczu i języku, na który jest tłumaczone dane szkolenie oraz dacie rozpoczęcia i zakończenia i limicie uczestników do tabeli courses oraz products

```
CREATE PROCEDURE [dbo].[uspAddCourse]
    @language_id int,
    @academic_id int,
    @interpreter_id int=null,
    @translated_to_id int=null,
    @course_name nvarchar(50),
    @start_date date,
    @end_date date,
    @participants_limit int
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Academics
            WHERE academic_id=@academic_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego nauczyciela!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @language_id=language_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @translated_to_id=language_id
        ) AND @translated_to_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Interpreters
            WHERE interpreter_id=@interpreter_id
        ) AND @interpreter_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego tłumacza!',1
        END

        DECLARE @type_id INT
        SELECT @type_id = product_type_id
        FROM ProductType
        WHERE 'course' = product_type_name


        BEGIN TRANSACTION
            INSERT INTO Products (product_type_id,language,academic_id,interpreter_id,translated_to)

                    values(@type_id,@language_id,@academic_id,@interpreter_id,@translated_to_id)

            DECLARE @product_id INT;
            SET  @product_id= SCOPE_IDENTITY();

            INSERT INTO Courses(product_id,course_name, start_date,end_date,participants_limit)
            Values (@product_id,@course_name, @start_date,@end_date,@participants_limit);
        COMMIT

    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd dodania kursu: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

setCoursePrice

Ustawia cenę zaliczki i/lub pełną cenę kursu

```
CREATE PROCEDURE [dbo].[uspSetCoursePrice]
    @course_id int,
    @advance_price money=null,
    @full_price money=null
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM courses
            where @course_id=product_id
        )
        BEGIN
            ;
            THROW 52000, N'Kurs o tej nazwie nie istnieje',1
        END


        IF @advance_price is not null
        Begin
            UPDATE courses
            SET advance_price=@advance_price
            where product_id=@course_id
        end

        IF @full_price is not null
        begin
            UPDATE courses
            SET full_price=@full_price
            where product_id=@course_id
        end
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany ceny kursu: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddStudies

Dodaje studia o podanej nazwie, id nauczyciela, nazwie języka oraz opcjonalnie danych o tłumaczu i języku, na który jest tłumaczone dane szkolenie oraz limicie uczestników do tabeli studies oraz products

```
CREATE PROCEDURE [dbo].[uspAddStudies]
    @language_id int,
    @academic_id int,
    @interpreter_id int=null,
    @translate_to_id int=null,
    @name nvarchar(50),
    @participants_limit int
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Academics
            WHERE academic_id=@academic_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego nauczyciela!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @language_id=language_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @translate_to_id=language_id
        ) AND @translate_to_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Interpreters
            WHERE interpreter_id=@interpreter_id
        ) AND @interpreter_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego tłumacza!',1
        END

        DECLARE @type_id INT
        SELECT @type_id = product_type_id
        FROM ProductType
        WHERE 'studies' = product_type_name

        BEGIN TRANSACTION
            INSERT INTO Products (product_type_id,language,academic_id,interpreter_id,translated_to)

                    values(@type_id,@language_id,@academic_id,@interpreter_id,@translate_to_id)

            DECLARE @product_id INT;
            SET  @product_id= SCOPE_IDENTITY();
            INSERT INTO Studies(product_id,name,participants_limit)
            Values (@product_id,@name,@participants_limit);
        COMMIT TRANSACTION

    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd dodania studiów: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

setStudiesPrice

Ustawia cenę zaliczki i/lub pełną cenę studiów o podanej nazwie

```
CREATE PROCEDURE [dbo].[uspSetStudiesPrice]
    @studies_id int,
    @advance_price money=null,
    @full_price money=null
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM studies
            where @studies_id=product_id
        )
        BEGIN
            ;
            THROW 52000, N'Studia o tej nazwie nie istnieją',1
        END

        IF @advance_price is not null
        Begin
            UPDATE studies
            SET advance_price=@advance_price
            where product_id=@studies_id
        end

        IF @full_price is not null
        begin
            UPDATE studies
            SET full_price=@full_price
            where product_id=@studies_id
        end


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany ceny studiów: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddStudiesMeetings

Dodaje spotkanie o podanej nazwie, id nauczyciela, nazwie języka oraz opcjonalnie danych o tłumaczu i języku, na który jest tłumaczone dane szkolenie oraz limicie uczestników, dacie spotkania i przynależności do danych studiów do tabeli StudiesMeetings oraz products

```
CREATE PROCEDURE [dbo].[uspAddStudiesMeetings]
    @language_id int,
    @academic_id int,
    @interpreter_id int=null,
    @translate_to_id int=null,
    @participants_limit int,
    @type_meeting_id INT,
    @date date,
    @studies_id int,
    @meeting_topic nvarchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Academics
            WHERE academic_id=@academic_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego nauczyciela!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @language_id=language_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Languages
            WHERE @translate_to_id=language_id
        ) AND @translate_to_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego języka!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Interpreters
            WHERE interpreter_id=@interpreter_id
        ) AND @interpreter_id is not null
        BEGIN
            ;
            THROW 52000, N'Nie ma takiego tłumacza!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM MeetingType
            WHERE type_name=@type_meeting_id
        )
        BEGIN
            ;
            THROW 52000, N'!Nie ma takiego typu spotkania!',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Studies
            WHERE product_id=@studies_id
        )
        BEGIN
            ;
            THROW 52000, N'Nie ma takich studiów',1
        END

        if @meeting_topic IS NULL
            BEGIN
                ;
                THROW 52000, N'Temat spotkania nie może być pusty!',1
            END


        DECLARE @type_id INT
        SELECT @type_id = product_type_id
        FROM ProductType
        WHERE 'meeting' = product_type_name
        BEGIN TRANSACTION
            INSERT INTO Products (product_type_id,language,academic_id,interpreter_id,translated_to)

                    values(@type_id,@language_id,@academic_id,@interpreter_id,@translate_to_id)

            DECLARE @product_id INT;
            SET  @product_id= SCOPE_IDENTITY();

            INSERT INTO StudiesMeetings(meeting_id,studies_id,date,type_id,participants_limit, meeting_topic)
            Values (@product_id,@studies_id,@date,@type_meeting_id,@participants_limit, @meeting_topic);
        COMMIT TRANSACTION

    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd dodania spotkania: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

SetMeetingPrice

Ustawia cenę danego spotkania dla studentów i/lub uczestników spoza studiów

```
CREATE PROCEDURE [dbo].[uspSetMeetingPrice]
    @meeting_id int,
    @student_price money=null,
    @outer_participant_price money=null
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM StudiesMeetings
            where @meeting_id=meeting_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki meeting nie istnieje',1
        END

        IF @student_price is not null
        Begin
            UPDATE StudiesMeetings
            SET student_price=@student_price
            where meeting_id=@meeting_id
        end

        IF @outer_participant_price is not null
        begin
            UPDATE StudiesMeetings
            SET outer_participant_price=@outer_participant_price
            where meeting_id=@meeting_id
        end

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany ceny spotkania: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddMeetingParticipant

Dodaje uczestnika spotkania o podanym id

```
CREATE PROCEDURE [dbo].[uspAddMeetingParticipant]
    @client_id int,
    @product_id int
```

```
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM Clients
            where @client_id=client_id
        )
        BEGIN
            ;
            THROW 52000, N'Klient o podanym id nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM StudiesMeetings
            where @product_id=meeting_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki meeting nie istnieje',1
        END

        DECLARE @participant_id INT;
        SELECT @participant_id=participant_id
        from StudiesParticipants
        where @client_id=client_id

        DECLARE @student_studies_id INT;
        SELECT @student_studies_id=product_id
        from StudiesParticipants
        where @client_id=client_id

        DECLARE @meeting_studies_id INT;
        SELECT @meeting_studies_id=studies_id
        from StudiesMeetings
        where meeting_id=@product_id

        IF @participant_id is null or @student_studies_id!=@meeting_studies_id
            BEGIN
                INSERT INTO OuterMeetingParticipants(client_id,meeting_id)
                values(@client_id,@product_id)
            END
        ELSE
            BEGIN
                INSERT INTO StudiesMeetingParticipants(meeting_id,participant_id)
                values(@product_id,@participant_id)
            END



    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd dodania uczestnika spotkania: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddWCSParticipant

Dodaje uczestnika do szkolenia podanego typu (kurs, studia, webinar)

```
CREATE PROCEDURE [dbo].[uspAddWCSParticipant]
    @type_id int,
    @client_id int,
    @product_id int

AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        DECLARE @type_name  nvarchar(50)
        SELECT @type_name = product_type_name
        FROM ProductType
        WHERE @type_id = product_type_id

        IF NOT EXISTS(
            SELECT *
            FROM ProductType
            where @type_name=product_type_name
        )
        BEGIN
            ;
            THROW 52000, N'Taki rodzaj szkolenia nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Clients
            where @client_id=client_id
        )
        BEGIN
            ;
            THROW 52000, N'Klient o podanym id nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM Products
            where @product_id=product_id and @type_id=product_type_id
        )
        BEGIN
            ;
            THROW 52000, N'Produkt nie istnieje lub jest innego typu niż podany',1
        END


        IF @type_name='webinars'
        begin
            INSERT INTO WebinarParticipants(product_id,client_id)
            values(@product_id,@client_id)
        end
        else IF @type_name='course'
        begin
            INSERT INTO CoursesParticipants(product_id,client_id)
            values(@product_id,@client_id)
        end
        else IF @type_name='studies'
        begin
            INSERT INTO StudiesParticipants(product_id,client_id)
            values(@product_id,@client_id)
        end
        else if @type_name='meeting'
        begin
            exec uspAddMeetingParticipant @client_id,@product_id
        end


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd dodania uczestnika: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

CancelPayment

Dla danego payment_id ustawia pole cancelled w tabeli Payments na true - anuluje płatność

```
CREATE PROCEDURE [dbo].[uspCancelPayment]
    @payment_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
```

```
        IF NOT EXISTS(
            SELECT *
            FROM Payments
            where @payment_id=payment_id
        )
        BEGIN
            ;
            THROW 52000, N'Płatność o podanym id nie istnieje',1
        END

        UPDATE Payments
        SET cancelled=1
        where payment_id=@payment_id

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd anulowania płatności: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

LetPayDaysLater

Zezwala użytkownikowi o podanym id na płacenie z podanym opóźnieniem (wartość w dniach)

```
CREATE PROCEDURE [dbo].[uspLetPayDaysLater]
    @client_id int,
    @days int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY


        IF NOT EXISTS(
            SELECT *
            FROM Clients
            where @client_id=client_id
        )
        BEGIN
            ;
            THROW 52000, N'Klient o podanym id nie istnieje',1
        END

        UPDATE Clients
        SET can_pay_days_later=@days
        where client_id=@client_id

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zezwolenia na opóźnienie w płatności: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

SetParticipantsLimit

Ustawia limit uczestników dla produktu podanego typu produktu (spotkania, kursu lub studiów)

```
CREATE PROCEDURE [dbo].[uspSetParticipantsLimit]
    @product_id int,
    @limit int,
    @product_type_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM ProductType
            where @product_type_id=product_type_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki rodzaj szkolenia nie istnieje',1
        END

        DECLARE @product_type_name nvarchar(50);
        Select @product_type_name=product_type_name
        from ProductType
        where @product_id=product_type_id

        IF @product_type_name='course'
        begin
            IF NOT EXISTS(
                SELECT *
                FROM Courses
                where @product_id=product_id
            )
            BEGIN
                ;
                THROW 52000, N'Taki kurs nie istnieje',1
            END

            UPDATE Courses
            SET participants_limit=@limit
            where product_id=@product_id
        end
        else IF @product_type_name='studies'
        begin
            IF NOT EXISTS(
                SELECT *
                FROM Studies
                where @product_id=product_id
            )
            BEGIN
                ;
                THROW 52000, N'Takie studia nie istnieją',1
            END

            UPDATE Studies
            SET participants_limit=@limit
            where product_id=@product_id
        end
        else IF @product_type_name='meeting'
        begin
            IF NOT EXISTS(
                SELECT *
                FROM StudiesMeetings
                where @product_id=meeting_id
            )
            BEGIN
                ;
                THROW 52000, N'Takie spotkanie nie istnieje',1
            END

            UPDATE StudiesMeetings
            SET participants_limit=@limit
            where meeting_id=@product_id
        end
        else
        BEGIN
            ;
            THROW 52000, N'Na podanym rodzaju szkolenia nie obowiązuje limit miejsc',1
        END

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany limitu miejsc: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
```

```
END
```

AddMeetingPresence

Dodaje status obecności na spotkaniu dla podanego użytkownika oraz id spotkania.

```
CREATE PROCEDURE [dbo].[uspAddMeetingPresence]
    @product_id int,
    @participant_id int,
    @presence bit
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM StudiesMeetings
            where @product_id=meeting_id
        )
        BEGIN
            ;
            THROW 52000, N'Takie spotkanie nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM StudiesParticipants
            where @participant_id=participant_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki uczestnik nie istnieje',1
        END

        INSERT INTO StudiesMeetingParticipants(participant_id,meeting_id,presence)
        values(@participant_id,@product_id,@presence)


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd wpisywania obecności na spotkaniu: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

SetMeetingPresence

Zmienia status obecności danego użytkownika na spotkaniu.

```
CREATE PROCEDURE [dbo].[uspSetMeetingPresence]
    @product_id int,
    @participant_id int,
    @presence bit
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM StudiesMeetings
            where @product_id=meeting_id
        )
        BEGIN
            ;
            THROW 52000, N'Takie spotkanie nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM StudiesParticipants
            where @participant_id=participant_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki uczestnik nie istnieje',1
        END

        UPDATE StudiesMeetingParticipants
        SET presence=@presence
        where @participant_id=participant_id and @product_id=meeting_id


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany obecności na spotkaniu: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddModulePresence

Dodaje status obecności na module dla podanego użytkownika oraz id modułu.

```
CREATE PROCEDURE [dbo].[uspAddModulePresence]
    @module_id int,
    @participant_id int,
    @presence bit
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Modules
            where @module_id=module_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki moduł nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM StudiesParticipants
            where @participant_id=participant_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki uczestnik nie istnieje',1
        END
        INSERT INTO ModulesAttendance(participant_id,module_id,presence)
        values(@participant_id,@module_id,@presence)


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd wpisywania obecności na module: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

SetModulePresence

Zmienia status obecności danego użytkownika na module.

```
CREATE PROCEDURE [dbo].[uspSetModulePresence]
    @module_id int,
    @participant_id int,
    @presence bit
AS
```

```
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Modules
            where @module_id=module_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki moduł nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM StudiesParticipants
            where @participant_id=participant_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki uczestnik nie istnieje',1
        END

        UPDATE ModulesAttendance
        SET presence=@presence
        where @participant_id=participant_id and @module_id=module_id


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany obecności na module: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddExamResult

Dodaje wynik egzaminu po podaniu przez użytkownika id egzaminu, id uczestnika studiów i punktów przez niego zdobytych

```
CREATE PROCEDURE [dbo].[uspAddExamResult]
    @exam_id int,
    @participant_id int,
    @points int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM Exams
            where @exam_id=exam_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki egzamin nie istnieje',1
        END

        IF NOT EXISTS(
            SELECT *
            FROM StudiesParticipants
            where @participant_id=participant_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki uczestnik studiów nie istnieje',1
        END

        DECLARE @max_points INT;
        SELECT @max_points = max_points
        FROM Exams
        WHERE exam_id=@exam_id
        IF @max_points<@points
        Begin
            ;
            THROW 52000, N'Liczba punktów przekracza wartość maksymalną',1
        END

        INSERT INTO ExamsTaken(exam_id,participant_id,points)
        values(@exam_id,@participant_id,@points)


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd wpisywania wyniku egzaminu: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddApprenticeship

Dla podanego uczestnika studiów dodaje datę odbycia przez niego praktyk do tabeli Apprenticeship

```
CREATE PROCEDURE [dbo].[uspAddApprenticeship]
    @date date,
    @participant_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM StudiesParticipants
            where @participant_id=participant_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki uczestnik studiów nie istnieje',1
        END


        IF GETDATE()<@date
        Begin
            ;
            THROW 52000, N'Wprowadzenie praktyk o dacie przyszłej niemożliwe',1
        END

        INSERT INTO Apprenticeship(participant_id,date)
        values(@participant_id,@date)


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd dodania praktyk: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

ChangeMeetingDate

Zmienia datę spotkania

```
CREATE PROCEDURE [dbo].[uspChangeMeetingDate]
    @meeting_id int,
    @date date
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM StudiesMeetings
            where @meeting_id=meeting_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki meeting nie istnieje',1
        END

        DECLARE @former_date DATE;
        SELECT @former_date=date
        FROM StudiesMeetings
        WHERE meeting_id=@meeting_id

        IF @former_date<GETDATE()
        Begin
            ;
            THROW 52000, N'Spotkanie się już odbyło - nie można zmienić jego daty!',1
        END

        IF @date<GETDATE()
        Begin
            ;
            THROW 52000, N'Data spotkania może być zmieniona tylko na przyszłą',1
        END


        UPDATE StudiesMeetings
        SET date=@date
        where meeting_id=@meeting_id

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany daty spotkania: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

DeleteProduct

Usuwa produkt o podanym id z bazy

```
CREATE PROCEDURE [dbo].[uspDeleteProduct]
    @product_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY
        IF NOT EXISTS(
            SELECT *
            FROM Products
            where @product_id=product_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki produkt nie istnieje',1
        END

        DELETE FROM Products Where @product_id=product_id

    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd usuwania produktu: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

Pay

Dla podanego order_id sumuje ceny produktów wyszczególnionych w order_details i dodaje do płatność do tabeli Payments oraz uczestników do tabel odpowiadających opłaconym szkoleniom

```
CREATE PROCEDURE [dbo].[uspPay]
    @order_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM Orders
            where @order_id=order_id
        )
        BEGIN
            ;
            THROW 52000, N'Takie zamówienie nie istnieje',1
        END

        Declare @total_price money;
        SET @total_price=0;

        DECLARE @client_id INT
        SELECT @client_id=client_id
        from orders
        where order_id=@order_id

        DECLARE @status INT;
        SELECT @status=status_id
        from statuses s
        join orders o
        on o.payment_status=s.status_id
        where order_id=@order_id

        DECLARE @initial_status INT;
        SET @initial_status=@status

        IF @initial_status=1
        BEGIN
            ;
            THROW 52000, N'Zamówienie było już opłacone',1
        END

        SELECT @status=status_id
        from statuses
        where status_name='paid'

        print(@status)

        DECLARE curOrder cursor for
        select product_id
        from Order_details
        where order_id=@order_id

        BEGIN TRANSACTION

            DECLARE @product_id INT;

            Open curOrder

            FETCH NEXT FROM curOrder INTO @product_id
            WHILE @@FETCH_STATUS = 0
            BEGIN
                DECLARE @is_advance bit
                SELECT @is_advance=is_advance
                from Order_details
                where @product_id=product_id and @order_id=order_id


                DECLARE @product_type nvarchar(50)
                SELECT @product_type=product_type_name
                from Products p
                join ProductType pt on pt.product_type_id=p.product_type_id
                where product_id=@product_id
                DECLARE @price money

                IF @product_type='webinar'
                BEGIN
                    select @price=price
                    from webinars
                    where @product_id=product_id

                END

                else IF @product_type='course'
                BEGIN
                    IF @is_advance=1
                    begin
                        SELECT @status=status_id
                        from statuses
                        where status_name='partially_paid'

                        select @price=advance_price
                        from courses
                        where @product_id=product_id
                    end
                    ELSE
                    begin
                        select @price=full_price
                        from courses
                        where @product_id=product_id
                    end
                END

                else IF @product_type='studies'
                BEGIN
                    IF @is_advance=1
                    begin
                        SELECT @status=status_id
                        from statuses
                        where status_name='partially_paid'


                        select @price=advance_price
                        from Studies
                        where @product_id=product_id
                    end
                    ELSE
                    begin
                        select @price=full_price
                        from studies
                        where @product_id=product_id
                    end

                END
                else if @product_type='meeting'
                begin

                    DECLARE @meeting_studies_id1 INT;
                    SELECT @meeting_studies_id1=studies_id
                    from StudiesMeetings
                    where meeting_id=@product_id
                    if exists(
                        Select *
                        from StudiesParticipants
                        where @client_id=client_id and product_id=@meeting_studies_id1
                    )
                    begin
                        select @price=student_price
                        from StudiesMeetings
                        where @product_id=meeting_id

                    end
                    else
                    begin
                        select @price=outer_participant_price
                        from StudiesMeetings
                        where @product_id=meeting_id

                    end
                end

                SET @total_price = @total_price +@price;

                FETCH NEXT FROM curOrder INTO @product_id;

            END

            close curOrder
            DEALLOCATE curOrder;


            IF @initial_status=(
            select status_id
            from Statuses
            where status_name='partially_paid'
            )
            Begin
                declare @former_price money;
                set @former_price=(select sum(price)
                from payments
                where order_id=@order_id
                group by order_id)
                print(@total_price)

                set @total_price=@total_price-@former_price
            end

            IF @total_price<0
            BEGIN
                ROLLBACK;
                THROW 52000, N'Cena ujemna!',1
            END

            INSERT INTO Payments(order_id,payment_date, price)
            Values (@order_id,GETDATE(),@total_price);

            UPDATE Orders
            SET payment_status=@status
            where order_id=@order_id

            --insert to relevant tables

            DECLARE curOrder1 cursor for
            select product_id
            from Order_details
            where order_id=@order_id



            Open curOrder1

            FETCH NEXT FROM curOrder1 INTO @product_id
            WHILE @@FETCH_STATUS = 0
            BEGIN

                DECLARE @product_type1 nvarchar(50)
                SELECT @product_type1=product_type_name
                from Products p
                join ProductType pt on pt.product_type_id=p.product_type_id
                where product_id=@product_id
                PRINT(@product_type1)


                IF @product_type1='webinar'
                BEGIN
                    if not exists(
                    select *
                    from WebinarParticipants
                    where @client_id=client_id
                    )
                    begin
                        Insert into WebinarParticipants(product_id,client_id)
                        values (@product_id,@client_id)
                    end
                END
                else IF @product_type1='course'
                BEGIN
                    if not exists(
                    select *
                    from CoursesParticipants
                    where @client_id=client_id
                    )
                    begin

                        Insert into CoursesParticipants(product_id,client_id)
                        values (@product_id,@client_id)
                    end
                END

                else IF @product_type1='studies'
                BEGIN
                    if not exists(
                    select *
                    from StudiesParticipants
                    where @client_id=client_id and @product_id=product_id
                    )
                    begin

                        Insert into StudiesParticipants(product_id,client_id)
                        values (@product_id,@client_id)
                    end
                END
                else if @product_type1='meeting'
                begin
                    DECLARE @meeting_studies_id INT;
                    SELECT @meeting_studies_id=studies_id
                    from StudiesMeetings
                    where meeting_id=@product_id
                    if exists(
                        Select *
                        from StudiesParticipants
                        where @client_id=client_id and product_id=@meeting_studies_id
                    )
                    begin
                        DECLARE @participant_id int
                        select @participant_id=participant_id
                        from StudiesParticipants
                        where client_id=@client_id
                        if not exists(
                        select *
                        from StudiesMeetingParticipants
                        where @participant_id=participant_id
                        )
                        begin
                            Insert into StudiesMeetingParticipants(meeting_id,participant_id,presence)
                            values (@product_id,@participant_id,0)
                        end
                    end
                    else
                    begin
                        if not exists(
                        select *
                        from OuterMeetingParticipants
                        where @client_id=client_id
                        )
                        begin
                            Insert into OuterMeetingParticipants(meeting_id,client_id,presence)
                            values (@product_id,@client_id,0)
                        end
                    end
                end


                FETCH NEXT FROM curOrder1 INTO @product_id;

            END

            close curOrder1
            DEALLOCATE curOrder1;
        COMMIT TRANSACTION

    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd płatności: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddOrder

Tworzy zamówienie dla klienta o podanym id

```
CREATE PROCEDURE [dbo].[uspAddOrder]
    @client_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM Clients
            where @client_id=client_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki klient nie istnieje',1
        END


        INSERT INTO Orders(client_id)
        values(@client_id)


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd tworzenia nowego zamówienia: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddProductToOrder

Dodaje produkt do podanego zamówienia oraz informację, czy jest to zaliczka czy nie

```
CREATE PROCEDURE [dbo].[uspAddProductToOrder]
    @order_id int,
    @product_id int,
    @is_advance bit
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM orders
            where @order_id=order_id
        )
        BEGIN
            ;
            THROW 52000, N'Takie zamówienie nie istnieje',1
        END

        IF NOT EXISTS(
            SELECT *
            FROM products
            where @product_id=product_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki produkt nie istnieje',1
        END

        IF @is_advance=1
        begin
            DECLARE @product_type nvarchar(50)
            select @product_type=product_type_name
            from Products p
            join ProductType pt
            on pt.product_type_id=p.product_type_id
            where @product_id=product_id

            if @product_type!='studies' and @product_type!='course'
            begin
                ;
                THROW 52000, N'Ten produkt nie posaida opcji "zaliczka"',1
            end

        end


        declare @status nvarchar(50)
        select @status=status_name
        from Statuses s
        join orders o
        on o.payment_status=s.status_id
        where order_id=@order_id

        IF @status!='not_paid'
        BEGIN
            ;
            THROW 52000, N'Nie można dodać produktu do zamówienia, którego płatność zaczęła być realizowana',1
        END


        INSERT INTO Order_details(order_id,product_id,is_advance)
        values(@order_id,@product_id,@is_advance)


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd dodawania produktu zamówienia: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

ChangeToFullPrice

Zmienia pole is_advance tabeli Order_details na false - oznacza to, że klient chce zapłacić pełną cenę po uprzednim zapłaceniu zaliczki

```
CREATE PROCEDURE [dbo].[uspChangeToFullPrice]
    @order_id int,
    @product_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM orders
            where @order_id=order_id
        )
        BEGIN
            ;
            THROW 52000, N'Takie zamówienie nie istnieje',1
        END
        IF NOT EXISTS(
            SELECT *
            FROM products
            where @product_id=product_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki produkt nie istnieje',1
        END


        Update Order_details
        set is_advance=0
        where order_id=@order_id and product_id=@product_id


    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd zmiany zaliczki na pełną cenę: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

DeleteProductFromOrder

Usuwa produkt z zamówienia

```
CREATE PROCEDURE [dbo].[uspDeleteProductFromOrder]
    @order_id int,
    @product_id int
AS
BEGIN

    SET NOCOUNT ON;
    BEGIN TRY

        IF NOT EXISTS(
            SELECT *
            FROM orders
            where @order_id=order_id
        )
        BEGIN
            ;
            THROW 52000, N'Takie zamówienie nie istnieje',1
        END

        IF NOT EXISTS(
            SELECT *
            FROM Order_details
            where @product_id=product_id and @order_id=order_id
        )
        BEGIN
            ;
            THROW 52000, N'Taki produkt nie istnieje w podanym zamówieniu',1
        END

        declare @status nvarchar(50)
        select @status=status_name
        from Statuses s
        join orders o
        on o.payment_status=s.status_id
        where order_id=@order_id

        IF @status!='not_paid'
        BEGIN
            ;
            THROW 52000, N'Nie można usunąć produktu z zamówienia, którego płatność zaczęła być realizowana',1
        END

        DELETE FROM Order_details
        where product_id=@product_id and order_id=@order_id
    END TRY
    BEGIN CATCH
        DECLARE @msg nvarchar(2048)=N'Błąd usunięcia produktu z zamówienia: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

AddParticipantAboveLimit

Dodaje uczestnika pomimo wyczerpania limitu uczestników

```
CREATE PROCEDURE [dbo].[uspAddParticipantAboveLimit]
    @type_id INT,
    @client_id int,
    @product_id int
AS
BEGIN
    IF NOT EXISTS(
        SELECT *
        FROM Clients
        where @client_id=client_id
    )
    BEGIN
        ;
        THROW 52000, N'Taki klient nie istnieje',1
    END
    IF NOT EXISTS(
        SELECT *
        FROM Products
        where @product_id=product_id
    )
    BEGIN
        ;
        THROW 52000, N'Taki produkt nie istnieje',1
    END
    IF NOT EXISTS(
        SELECT *
        FROM ProductType
        where @type_id=product_type_id
    )
    BEGIN
        ;
        THROW 52000, N'Taki typ produktu nie istnieje',1
    END
    SET NOCOUNT ON;
    BEGIN TRY
        DISABLE TRIGGER checkStudiesParticipantsLimit_trg on StudiesParticipants;
        DISABLE TRIGGER checkStudiesMeetingLimit_studiesParticipants_trg on StudiesMeetingParticipants;
        DISABLE TRIGGER checkStudiesMeetingLimit_outerParticipants_trg on OuterMeetingParticipants;
        DISABLE TRIGGER checkCourseParticipantsLimit_trg on CoursesParticipants;
        DISABLE TRIGGER checkIfClientPaidForStudies_outerParticipant_trg on OuterMeetingParticipants;
        DISABLE TRIGGER checkIfClientPaidForStudiesMeeting_outerParticipant_trg on OuterMeetingParticipants;
        DISABLE TRIGGER checkIfClientPaidForCourse_trg on CoursesParticipants;

        begin transaction
            exec uspAddWCSParticipant @type_id,@client_id,@product_id;

            INSERT INTO Orders(client_id)
            values(@client_id)

            DECLARE @order_id INT;
            SET  @order_id= SCOPE_IDENTITY();
            PRINT(@order_id)

            DECLARE @is_advance BIT;
            IF @type_id=2 or @type_id=3
            BEGIN
                SET @is_advance=1
            END
            ELSE
            BEGIN
                SET @is_advance=0
            END

            exec uspAddProductToOrder @order_id,@product_id,@is_advance;
        commit transaction;

        ENABLE TRIGGER checkStudiesParticipantsLimit_trg on StudiesParticipants;
        ENABLE TRIGGER checkStudiesMeetingLimit_studiesParticipants_trg on StudiesMeetingParticipants;
        ENABLE TRIGGER checkStudiesMeetingLimit_outerParticipants_trg on OuterMeetingParticipants;
        ENABLE TRIGGER checkCourseParticipantsLimit_trg on CoursesParticipants;
        ENABLE TRIGGER checkIfClientPaidForStudies_outerParticipant_trg on OuterMeetingParticipants;
        ENABLE TRIGGER checkIfClientPaidForStudiesMeeting_outerParticipant_trg on OuterMeetingParticipants;
        ENABLE TRIGGER checkIfClientPaidForCourse_trg on CoursesParticipants;


    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRAN
        DECLARE @msg nvarchar(2048)=N'Błąd dodania uczestnika ponad limit: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END
```

Funkcje

Ogólne

GetProductName

Umożliwia konwersję id produktu na nazwę, wykorzystywaną w innych funkcjach i widokach

```
CREATE FUNCTION getProductName(@product_id int)
    RETURNS nvarchar(50)
AS
    BEGIN
        DECLARE @product_type nvarchar(50)
        SET @product_type = ISNULL((SELECT product_type_name
                             FROM Products INNER JOIN ProductType ON Products.product_type_id =
ProductType.product_type_id
                             WHERE product_id = @product_id), 'Nan')

        RETURN CASE @product_type
                WHEN 'Nan' THEN ''
                WHEN 'webinar' THEN (SELECT webinar_name FROM Webinars WHERE product_id = @product_id)
                WHEN 'studies' THEN (SELECT name FROM Studies WHERE product_id = @product_id)
                WHEN 'meeting' THEN (SELECT meeting_topic FROM StudiesMeetings WHERE meeting_id = @product_id)
                WHEN 'course' THEN (SELECT course_name FROM Courses WHERE product_id = @product_id)
            END
    END
```

GetUserIdFromUserEmail

```
CREATE FUNCTION getUserIdFromUserEmail(@user_email nvarchar(50))
    RETURNS int
AS
    BEGIN
        DECLARE @user_id int
        SET @user_id = (SELECT user_id FROM Users WHERE email = @user_email)

        RETURN @user_id
    END
```

GetParticipantIdFromUserAndProduct

```
CREATE FUNCTION getParticipantIdFromUserAndProduct(@user_id int, @product_id int)
    RETURNS int
AS
    BEGIN
        DECLARE @product_type nvarchar(50)
        SET @product_type = (SELECT product_type_name
                             FROM Products
                                INNER JOIN ProductType ON Products.product_type_id =
ProductType.product_type_id
                             WHERE product_id = @product_id)

        RETURN CASE @product_type
            WHEN 'webinar' THEN (SELECT client_id
                                 FROM WebinarParticipants
                                 WHERE product_id = @product_id AND client_id = @user_id )
            WHEN 'course' THEN (SELECT participant_id
                                FROM CoursesParticipants
                                WHERE product_id = @product_id AND client_id = @user_id)
            WHEN 'studies' THEN (SELECT participant_id
                                 FROM StudiesParticipants
                                 WHERE product_id = @product_id AND client_id = @user_id)
            WHEN 'meeting' THEN (SELECT client_id
                                 FROM OuterMeetingParticipants
                                 WHERE meeting_id = @product_id AND client_id = @user_id)
        END
    END
```

CheckIfClientPaid

Sprawdza czy dany klient zapłacił za dany produkt

```
CREATE FUNCTION checkIfClientPaid(@client_id int, @product_id int)
    RETURNS bit
BEGIN
    DECLARE @payment_status nvarchar(50)
    SET @payment_status = ISNULL((SELECT status_name
                                  FROM Orders
                                           INNER JOIN Order_details ON Orders.order_id = Order_details.order_id
                                           INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
                                  WHERE
                                          Order_details.product_id = @product_id AND
                                          Orders.client_id = @client_id), 'none')

    IF @payment_status = 'paid' OR @payment_status = 'partially_paid' BEGIN
        RETURN 1
    END
    RETURN 0
END
```

Sekretarz

ClientsExams

Lista wyników egzaminów dla danego klienta

```
CREATE FUNCTION clientsExam(@participant_id int)
    RETURNS table
        AS
        RETURN Select e.studies_id, et.exam_id, et.points
        FROM Exams as e
        inner join ExamsTaken as et on et.exam_id=e.exam_id and et.participant_id = @participant_id
```

ClientsApprenticeships

Liczba odbytych praktyk przez danego klienta

```
CREATE FUNCTION clientsApprenticeships(@participant_id int)
    RETURNS int
AS
BEGIN
    RETURN ( SELECT COUNT(date) FROM Apprenticeship
    Where participant_id = @participant_id
    Group By participant_id)
END
```

Kursy

CoursePass

Wypisanie wartości 1 gdy uczestnik zaliczył kurs i 0 gdy nie zaliczył

```
CREATE FUNCTION coursePass(@participant_id int)
    RETURNS bit
AS
BEGIN
    DECLARE @course_id int
    SET @course_id = (Select product_id
                        FROM CoursesParticipants
                        WHERE @participant_id=participant_id)
    DECLARE @presence float
    SET @presence = [dbo].[coursesPresence](@participant_id, @course_id)
    DECLARE @pass bit
    SET @pass = IIF(@presence >= 80, 1, 0)
    RETURN @pass
END
```

CourseInfo

Wypisanie podstawowych informacji o kursie takich jak: nazwa, cena, zaliczka, data rozpoczecia, data zakonczenia oraz język główny i jezyk na który kurs jest tłumaczony.

```
CREATE FUNCTION courseInfo(@product_id int)
    RETURNS table
        AS
        RETURN Select c.course_name as course_name,
        c.full_price as price,
        c.advance_price as advance_price,
        c.start_date as start_date,
        c.end_date as end_date,
        p.language as orginal_language,
        l.language_name as translated_to
        FROM Products as p
        join Courses as c on c.product_id=p.product_id
        left outer join Languages as l on l.language_id=p.translated_to
        WHERE p.product_id=@product_id
```

ModulesPresence

Sprawdzenie statusu swojej obecności na wybranych modułach

```
CREATE FUNCTION modulesPresence(@participant_id int, @module_id int)
    RETURNS bit
AS
BEGIN
    DECLARE @presence BIT
    SET @presence = ISNULL((SELECT presence
                FROM ModulesAttendance
                WHERE participant_id=@participant_id AND
                module_id=@module_id),0)
    RETURN @presence
END
```

CoursesPresence

Sprawdzenie procentowej obecności na modułach w danym kursie

```
CREATE FUNCTION [dbo].[coursesPresence](@participant_id int, @product_id int)
    RETURNS FLOAT
AS
BEGIN
    DECLARE @presence float
    SET @presence = ISNULL((SELECT COUNT(ma.presence)
                            FROM ModulesAttendance as ma
                            inner join Modules as m
                                on m.module_id=ma.module_id and m.product_id = @product_id
                            WHERE ma.participant_id=@participant_id and ma.presence=1),0)
    DECLARE @modules_num int
    SET @modules_num = ISNULL((SELECT COUNT(module_id)
                    FROM Modules
                    WHERE product_id = @product_id),0)
    IF @modules_num = 0
        RETURN 100
    RETURN (@presence/@modules_num) *100
END
go
```

CoursesFreeSlots

Sprawdzenie ilości wolnych miejsc na kursach hybrydowych i stacjonarnych

```
CREATE FUNCTION coursesFreeSlots(@product_id int)
    RETURNS INT
AS
BEGIN
    DECLARE @slots INT
    SET @slots = ISNULL((SELECT c.participants_limit
                    From Courses as c
                    Where c.product_id = @product_id), 0)
    DECLARE @occupied INT
    SET @occupied = ISNULL((SELECT COUNT(cp.participant_id)
                From CoursesParticipants as cp
                WHERE cp.product_id = @product_id
                GROUP BY cp.product_id),0)
    RETURN @slots - @occupied
END
```

ClientsCourses

Sprawdzenie na jakie kursy jest zapisany dany klient oraz status płatności tego kursu

```
CREATE FUNCTION clientCourses(@client_id int)
    RETURNS table
        AS
        RETURN Select c.course_name, s.status_name
        FROM Orders as o inner join Order_details as od on od.order_id=o.order_id
        inner join Courses as c on c.product_id=od.product_id
        inner join Statuses as s on s.status_id=o.payment_status
        WHERE o.client_id=@client_id
```

CheckIfCourseParticipantsAllowed

Sprawdza czy do kursu można dopisać więcej osób - czy limit miejsc nie został jeszcze przekroczony

```
CREATE FUNCTION checkIfCourseParticipantsAllowed(@product_id int)
        RETURNS bit
    AS
        BEGIN
            DECLARE @participant_limit int
            DECLARE @participants_count int

            SET @participants_count = ISNULL((SELECT COUNT(*)
                                          FROM CoursesParticipants
                                          WHERE product_id = @product_id
                                          GROUP BY product_id), 0)

            SET @participant_limit = ISNULL((SELECT participants_limit
                                         FROM Courses
                                         WHERE product_id = @product_id), 0)
            IF @participants_count > @participant_limit BEGIN
                RETURN 0
            END
            RETURN 1
END
```

Studia

StudiesPass

Umożliwia sprawdzenie czy dany uczestnik studiów zaliczył studia - wystarczy jedynie participant_id, ponieważ jest on unikalny i identyfikuje danego klienta od razu w kontekście studiów.

```
CREATE FUNCTION studiesPass(@participant_id int)
    RETURNS bit
AS
    BEGIN
       IF dbo.checkApprenticeshipStatus(@participant_id) = 1 AND
          dbo.studiesPresence(@participant_id) >= 80 AND
          dbo.checkExamStatus(@participant_id) = 1
            RETURN 1
       RETURN 0
    END
```

StudiesPresence

Sprawdzenie obecności danego uczestnika studiów

```
CREATE FUNCTION studiesPresence(@participant_id int)
    RETURNS float
AS
    BEGIN
        DECLARE @meetingsCount int
        SET @meetingsCount = ISNULL((SELECT COUNT(*)
                                    FROM StudiesMeetings
                                      INNER JOIN StudiesMeetingParticipants ON StudiesMeetings.meeting_id =
StudiesMeetingParticipants.meeting_id
                                    WHERE date < GETDATE() AND participant_id = @participant_id), 0)
        IF @meetingsCount = 0 BEGIN
           RETURN 100
        END

        DECLARE @attendedMeetings int
        SET @attendedMeetings = ISNULL((SELECT COUNT(*)
                                         FROM StudiesMeetings
                                             INNER JOIN StudiesMeetingParticipants ON StudiesMeetings.meeting_id
= StudiesMeetingParticipants.meeting_id

                                         WHERE
                                             date < GETDATE() AND
                                             presence = 1 AND
                                             participant_id = @participant_id), 0)

        RETURN CAST(@attendedMeetings AS float)/@meetingsCount * 100.0
    END
go
```

GetExamScores

Umożliwia wyświetlenie punktów i wyniku procentowego z egzaminów w których uczestnik studiów brał udział (dla wszystkich studiów na które dany klient został zapisany)

```
CREATE FUNCTION getExamScores(@student_id int)
    RETURNS table
AS
    RETURN
        SELECT name, date, points, CAST(points AS float)/max_points*100 AS percentScore
        FROM ExamsTaken
            INNER JOIN Exams ON ExamsTaken.exam_id = Exams.exam_id
            INNER JOIN dbo.Studies S on Exams.studies_id = S.product_id
        WHERE participant_id = @student_id
```

CheckExamStatus

Umożliwia sprawdzenie czy dany uczestnik studiów zaliczył egzaminy

```
CREATE FUNCTION checkExamStatus(@participan_id int)
    RETURNS bit
AS
    BEGIN
        DECLARE @passed_exams_count int
        SET @passed_exams_count = ISNULL((SELECT COUNT(*)
                                          FROM dbo.getExamScores(@participan_id)
                                          WHERE percentScore >= 50), 0)
        IF @passed_exams_count >= 1
            RETURN 1
        RETURN 0
    END
go
```

CheckExamMaxPoints

Pozwala sprawdzić maksymalną ilość punktów na danym egzaminie

```
CREATE FUNCTION checkExamMaxPoints(@exam_id int)
    RETURNS int
AS
BEGIN
    DECLARE @exam_max_points int
    SET @exam_max_points = ISNULL((SELECT max_points
                             FROM Exams
                             WHERE exam_id = @exam_id), 0)
    RETURN @exam_max_points
END
go
```

CheckExamDate

Pozwala sprawdzić datę wybranego egzaminu

```
CREATE FUNCTION checkExamDate(@exam_id int)
    RETURNS date
AS
BEGIN
    DECLARE @exam_date date
    SET @exam_date = ISNULL((SELECT date
                                    FROM Exams
                                    WHERE exam_id = @exam_id), NULL)
    RETURN @exam_date
END
```

GetStudiesMeetings

Umożliwia wyświetlenie wszystkich zaplanowanych spotkań na studiach

```
CREATE FUNCTION getStudiesMeetings(@studies_id int)
    RETURNS table
AS RETURN
        SELECT meeting_topic, date, participants_limit
        FROM StudiesMeetings
        WHERE studies_id = @studies_id
        ORDER BY date
```

GetRegisteredApprenticeship

Umożliwia wyświetlenie praktyk danego uczestnika studiów

```
CREATE FUNCTION getRegisteredApprenticeship(@participant_id int)
    RETURNS table
AS RETURN
        SELECT name, Apprenticeship.*
        FROM Apprenticeship
            INNER JOIN StudiesParticipants ON Apprenticeship.participant_id =
StudiesParticipants.participant_id
            INNER JOIN Studies ON StudiesParticipants.product_id = Studies.product_id
        WHERE Apprenticeship.participant_id = @participant_id
```

CheckApprenticeshipStatus

Umożliwia sprawdzenie czy dany uczestnik studiów ma zaliczone praktyki

```
CREATE FUNCTION checkApprenticeshipStatus(@participant_id int)
    RETURNS bit
AS
    BEGIN
        DECLARE @acceptedApprenticeshipStatus int
        SET @acceptedApprenticeshipStatus = ISNULL((SELECT COUNT(*)
                                                    FROM Apprenticeship
                                                    WHERE presence_percentage = 100 AND participant_id =
@participant_id), 0)
        IF @acceptedApprenticeshipStatus >= 2
           RETURN 1

        RETURN 0
    END
go
```

CheckParticipantsLimit

Pozwala sprawdzić limit osób zapisanych na studiach

```
CREATE FUNCTION checkParicipantsLimit(@studies_id int)
    RETURNS int
AS
    BEGIN
        DECLARE @paricipantsLimit int
        SET @paricipantsLimit = ISNULL((SELECT participants_limit
                                        FROM Studies
                                        WHERE product_id = @studies_id), 0)
        RETURN @paricipantsLimit
    END
go
```

CheckIfStudiesMeetingParticipantsAllowed

Pozwala sprawdzić czy do listy uczestników spotkania na studiach można dopisać więcej osób

```
CREATE FUNCTION checkIfStudiesMeetingParticipantsAllowed(@meeting_id int)
    RETURNS bit
AS
    BEGIN
        DECLARE @outer_participant_count int
        DECLARE @studies_participant_count int
        DECLARE @participant_limit int

        SET @studies_participant_count = ISNULL((SELECT COUNT(*)
                                                 FROM StudiesMeetingParticipants
                                                 WHERE meeting_id = @meeting_id
                                                 GROUP BY meeting_id), 0)
        SET @outer_participant_count = ISNULL((SELECT COUNT(*)
                                               FROM OuterMeetingParticipants
                                               WHERE meeting_id = @meeting_id
```

```
                                             GROUP BY meeting_id), 0)
        SET @participant_limit = ISNULL((SELECT participants_limit
                                        FROM StudiesMeetings
                                        WHERE meeting_id = @meeting_id), 0)


        IF @studies_participant_count + @outer_participant_count > @participant_limit BEGIN
            RETURN 0
        END
        RETURN 1
    END
```

CheckIfStudiesParticipantsAllowed

Pozwala sprawdzić czy limit uczestników zapisanych na dane studia nie został przekroczony

```
CREATE FUNCTION checkIfStudiesParticipantsAllowed(@product_id int)
    RETURNS bit
AS
BEGIN
    DECLARE @participant_limit int
    DECLARE @participants_count int

    SET @participants_count = ISNULL((SELECT COUNT(*)
                                        FROM StudiesParticipants
                                        WHERE product_id = @product_id
                                        GROUP BY product_id), 0)

    SET @participant_limit = dbo.checkParicipantsLimit(@product_id)


    IF @participants_count > @participant_limit BEGIN
        RETURN 0
    END
    RETURN 1
END
```

Nauczyciel

GetTaughtWebinars

Umożliwia wyświetlenie prowadzonych przez nauczyciela webinarów

```

CREATE FUNCTION getTaughtWebinars(@academic_id int)
    RETURNS table
AS RETURN
    SELECT webinar_name, Webinars.product_id
    FROM Products
        INNER JOIN Webinars ON Products.product_id = Webinars.product_id
    WHERE academic_id = @academic_id

```

GetTaughtCourses

Umożliwia wyświetlenie prowadzonych przez nauczyciela kursów

```

CREATE FUNCTION getTaughtCourses(@academic_id int)
    RETURNS table
AS RETURN
    SELECT course_name, Courses.product_id
    FROM Products
        INNER JOIN Courses ON Products.product_id = Courses.product_id
    WHERE academic_id = @academic_id

```

GetTaughtMeetings

Umożliwia wyświetlenie prowadzonych przez nauczyciela kursów

```
CREATE FUNCTION getTaughtStudiesMeetings(@academic_id int)
    RETURNS table
AS RETURN
    SELECT meeting_topic, meeting_id
    FROM Products
        INNER JOIN StudiesMeetings ON Products.product_id = StudiesMeetings.meeting_id
    WHERE academic_id = @academic_id
```

GetTaughtStudies

Umożliwia wyświetlenie prowadzonych przez nauczyciela kursów

```
CREATE FUNCTION getTaughtStudies(@academic_id int)
    RETURNS table
AS RETURN
    SELECT name, Studies.product_id
    FROM Products
        INNER JOIN Studies ON Products.product_id = Studies.product_id
    WHERE academic_id = @academic_id
```

GetStudiesMeetingAttendanceList

Umożliwia wyswietlenie listy obecności na danym spotkaniu na studiach

```
CREATE FUNCTION getStudiesMeetingAttendanceList(@meeting_id int)
    RETURNS table
AS RETURN
    SELECT StudiesMeetingParticipants.participant_id, U.last_name, U.first_name
    FROM StudiesMeetingParticipants
        INNER JOIN dbo.StudiesMeetings SM on StudiesMeetingParticipants.meeting_id = SM.meeting_id
        INNER JOIN StudiesParticipants SP on StudiesMeetingParticipants.participant_id = SP.participant_id
        INNER JOIN Clients C on SP.client_id = C.client_id
        INNER JOIN Users U on C.client_id = U.user_id
    WHERE SM.meeting_id = @meeting_id
    UNION
    SELECT OuterMeetingParticipants.client_id, U.last_name, U.first_name
    FROM OuterMeetingParticipants
        INNER JOIN Clients C ON OuterMeetingParticipants.client_id = C.client_id
        INNER JOIN Users U ON C.client_id = U.user_id
    WHERE meeting_id = @meeting_id
go
```

GetCourseModuleAttendanceList

Wyświetla liste uczestników danego modułu z kursu

```
CREATE FUNCTION getCourseModuleAttendanceList(@module_id int)
    RETURNS table
AS RETURN
    SELECT ModulesAttendance.participant_id, last_name, first_name
    FROM ModulesAttendance
        INNER JOIN CoursesParticipants CP ON ModulesAttendance.participant_id = CP.participant_id
        INNER JOIN dbo.Clients C on C.client_id = CP.client_id
        INNER JOIN Users U on C.client_id = U.user_id
    WHERE module_id = @module_id
```

Klient

GetOwnedWebinars

Umożliwia wyświetlenie zakupionych webinarów przez klienta

```
CREATE FUNCTION [dbo].[getOwnedWebinars](@client_id int)
    RETURNS table
AS RETURN
    SELECT webinar_name
    FROM Webinars
        INNER JOIN Products ON Webinars.product_id = Products.product_id
        INNER JOIN Order_details ON Products.product_id = Order_details.product_id
        INNER JOIN Orders ON Order_details.order_id = Orders.order_id
        INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
        INNER JOIN Payments ON Payments.order_id=Orders.order_id
    WHERE status_name = 'paid' AND client_id = @client_id AND DATEDIFF(d,payment_date,GETDATE())<=30
```

GetOwnedStudies

Umożliwia wyświetlenie zakupionych studiów przez klienta

```
CREATE FUNCTION getOwnedStudies(@client_id int)
    RETURNS table
        AS RETURN
        SELECT name
        FROM Studies
                 INNER JOIN Products ON Studies.product_id = Products.product_id
                 INNER JOIN Order_details ON Products.product_id = Order_details.product_id
                 INNER JOIN Orders ON Order_details.order_id = Orders.order_id
                 INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
        WHERE status_name = 'paid' AND client_id = @client_id
```

GetOwnedStudiesMeetings

Umożliwia wyświetlenie zakupionych spotkań ze studiów przez klienta

```
CREATE FUNCTION getOwnedStudiesMeetings(@client_id int)
    RETURNS table
        AS RETURN
        SELECT meeting_topic
        FROM StudiesMeetings
                 INNER JOIN Products ON StudiesMeetings.meeting_id = Products.product_id
                 INNER JOIN Order_details ON Products.product_id = Order_details.product_id
                 INNER JOIN Orders ON Order_details.order_id = Orders.order_id
                 INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
        WHERE status_name = 'paid' AND client_id = @client_id
```

GetOwnedCourses

Umożliwia wyświetlenie zakupionych kursów przez klienta

```
CREATE FUNCTION getOwnedCourses(@client_id int)
    RETURNS table
        AS RETURN
        SELECT course_name
        FROM Courses
                 INNER JOIN Products ON Courses.product_id = Products.product_id
                 INNER JOIN Order_details ON Products.product_id = Order_details.product_id
                 INNER JOIN Orders ON Order_details.order_id = Orders.order_id
                 INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
        WHERE status_name = 'paid' AND client_id = @client_id
```

GetBucket

Pozwala wyświetlić zawartość koszyka klientów

```
CREATE FUNCTION getBucket(@client_id int)
    RETURNS table
```

```
AS RETURN
    SELECT dbo.getProductName(Products.product_id) AS product_name, product_type_name, Payments.price
    FROM Products
        INNER JOIN Order_details ON Products.product_id = Order_details.product_id
        INNER JOIN Orders ON Order_details.order_id = Orders.order_id
        INNER JOIN ProductType ON Products.product_type_id = ProductType.product_type_id
        INNER JOIN Payments ON Orders.order_id = Payments.order_id
        INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
    WHERE status_name = 'not_paid' AND client_id = @client_id
go
```

GetPaymentHistory

Umożliwia wyświetlenie historii płatności danego klienta

```
CREATE FUNCTION getPaymentHistory(@client_id int)
    RETURNS table
AS RETURN
    SELECT payment_date, price, Orders.order_id
    FROM Payments
        INNER JOIN Orders ON Payments.order_id = Orders.order_id
        INNER JOIN Statuses ON Orders.payment_status = Statuses.status_id
    WHERE status_name = 'paid' AND client_id = @client_id
go
```

Triggery

Studia

CheckStudiesMeetingLimit

Przy dodawaniu nowych uczestników spotkań sprawdza czy nie został przekroczony limit miejsc na spotkaniu na studiach podczas wpisywania do tabeli `StudiesMeetingParticipants` lub `OuterMeetingParticipants`

```
CREATE TRIGGER checkStudiesMeetingLimit_studiesParticipants_trg
ON StudiesMeetingParticipants
AFTER INSERT
AS
    BEGIN
        SET NOCOUNT ON
        DECLARE @meeting_id int
        DECLARE curs CURSOR FOR
            (SELECT meeting_id FROM inserted)

        OPEN curs

        FETCH NEXT FROM curs INTO @meeting_id
        WHILE @@FETCH_STATUS = 0 BEGIN
            IF NOT dbo.checkIfStudiesMeetingParticipantsAllowed(@meeting_id) = 1 BEGIN
                RAISERROR(N'Studies Meetings participants limit exceeded', 12, 1)
            END

            FETCH NEXT FROM curs INTO @meeting_id
        END
        CLOSE curs
        DEALLOCATE curs
    END

CREATE TRIGGER checkStudiesMeetingLimit_outerParticipants_trg
ON OuterMeetingParticipants
AFTER INSERT
AS
    BEGIN
        SET NOCOUNT ON
        DECLARE @meeting_id int
        DECLARE curs CURSOR FOR
            (SELECT meeting_id FROM inserted)

        OPEN curs
        FETCH NEXT FROM curs INTO @meeting_id
        WHILE @@FETCH_STATUS = 0 BEGIN
            IF NOT dbo.checkIfStudiesMeetingParticipantsAllowed(@meeting_id) = 1 BEGIN
                RAISERROR(N'Studies Meetings participants limit exceeded', 12, 1)
            END

            FETCH NEXT FROM curs INTO @meeting_id
        END
        CLOSE curs
        DEALLOCATE curs
    END
```

CheckStudiesParticipantsLimit

Przy dodawaniu nowych uczestników na studia do tabli `StudiesParticipants`, sprawdza czy limit zapisanych uczestników nie został przekroczony

```
CREATE TRIGGER checkStudiesParticipantsLimit_trg
    ON StudiesParticipants
    AFTER INSERT
    AS
BEGIN
    SET NOCOUNT ON
    DECLARE @studies_id int


    DECLARE curs CURSOR FOR
        (SELECT product_id FROM inserted)

    OPEN curs

    FETCH NEXT FROM curs INTO @studies_id
    WHILE @@FETCH_STATUS = 0 BEGIN
        IF NOT dbo.checkIfStudiesParticipantsAllowed (@studies_id) = 1 BEGIN
            RAISERROR(N'Studies Participants limit exceeded', 12, 1)
        END

        FETCH NEXT FROM curs INTO @studies_id
    END
    CLOSE curs
    DEALLOCATE curs
END
```

CheckIfClientPaidForStudies

Podczas wpisywania do tabeli `StudiesParticipants` sprawdza czy wpisywany klient zapłacił za studia

```
CREATE TRIGGER checkIfClientPaidForStudies_trg
        ON StudiesParticipants
        AFTER INSERT
AS
    BEGIN
        SET NOCOUNT ON
        DECLARE @client_id int
        DECLARE @product_id int

        DECLARE curs CURSOR FOR
            (SELECT client_id, product_id FROM inserted)

        OPEN curs

        FETCH NEXT FROM curs INTO @client_id, @product_id
        WHILE @@FETCH_STATUS = 0 BEGIN
            IF NOT dbo.checkIfClientPaid(@client_id, @product_id) = 1 BEGIN
                RAISERROR(N'Client did not pay for the product', 12, 1)
            END

            FETCH NEXT FROM curs INTO @client_id, @product_id
        END
        CLOSE curs
        DEALLOCATE curs
    END
```

CheckIfClientPaidForStudiesMeeting

Podczas wpisywania do tabeli `OuterMeetingParticipants` sprawdza czy wpisywani klienci mają status zamówienia jako zapłacony.

```sql
CREATE TRIGGER checkIfClientPaidForStudiesMeeting_outerParticipant_trg
    ON OuterMeetingParticipants
    AFTER INSERT
    AS
BEGIN
    SET NOCOUNT ON
    DECLARE @client_id int
    DECLARE @meeting_id int

    DECLARE curs CURSOR FOR
        (SELECT client_id, meeting_id FROM inserted)

    OPEN curs

    FETCH NEXT FROM curs INTO @client_id, @meeting_id
    WHILE @@FETCH_STATUS = 0 BEGIN
        IF NOT dbo.checkIfClientPaid(@client_id, @meeting_id) = 1 BEGIN
            RAISERROR(N'Client did not pay for the product', 12, 1)
        END

        FETCH NEXT FROM curs INTO @client_id, @meeting_id
    END
    CLOSE curs
    DEALLOCATE curs
END
```

Kursy

CheckCourseParticipantsLimit

Przy wpisywaniu do tabeli `CoursesParticipants` sprawdza czy limit uczestników zapisanych na kurs nie został przekroczony

```sql
CREATE TRIGGER checkCourseParticipantsLimit_trg
    ON CoursesParticipants
    AFTER INSERT
    AS
BEGIN
    SET NOCOUNT ON
    DECLARE @course_id int

    DECLARE curs CURSOR FOR
        (SELECT product_id FROM inserted)

    OPEN curs

    FETCH NEXT FROM curs INTO @course_id
    WHILE @@FETCH_STATUS = 0 BEGIN
        IF NOT dbo.checkIfCourseParticipantsAllowed(@course_id) = 1 BEGIN
            RAISERROR(N'Course Participants limit exceeded', 12, 1)
        END

        FETCH NEXT FROM curs INTO @course_id
    END
    CLOSE curs
    DEALLOCATE curs
END
```

CheckIfClientPaidForCourse

Przy wpisywaniu do tabeli `CoursesParticipants` sprawdza czy klient zapłacił za dany kurs

```sql
CREATE TRIGGER checkIfClientPaidForCourse_trg
    ON CoursesParticipants
    AFTER INSERT
    AS
```

```
BEGIN
    SET NOCOUNT ON
    DECLARE @client_id int
    DECLARE @product_id int

    DECLARE curs CURSOR FOR
        (SELECT client_id, product_id FROM inserted)

    OPEN curs

    FETCH NEXT FROM curs INTO @client_id, @product_id
    WHILE @@FETCH_STATUS = 0 BEGIN
        IF NOT dbo.checkIfClientPaid(@client_id, @product_id) = 1 BEGIN
            RAISERROR(N'Client did not pay for the product', 12, 1)
        END

        FETCH NEXT FROM curs INTO @client_id, @product_id
    END
    CLOSE curs
    DEALLOCATE curs
END
```

Webinary

Przy wpisywaniu do tabeli `WebinarParticipants` sprawdza czy klient zapłacił za webinar.

```
CREATE TRIGGER checkIfClientPaidForWebinar_trg
    ON WebinarParticipants
    AFTER INSERT
    AS
BEGIN
    SET NOCOUNT ON
    DECLARE @product_id int
    DECLARE @client_id int

    DECLARE curs CURSOR FOR
        (SELECT product_id, client_id FROM inserted)

    OPEN curs

    FETCH NEXT FROM curs INTO @product_id, @client_id
    WHILE @@FETCH_STATUS = 0 BEGIN
        IF NOT dbo.checkIfClientPaid(@client_id, @product_id) = 1 BEGIN
            RAISERROR(N'Client did not pay for the product', 12, 1)
        END

        FETCH NEXT FROM curs INTO @product_id, @client_id
    END
    CLOSE curs
    DEALLOCATE curs
END
```

Role i upoważnienia

Sekretarz

```
Create role secretary

GRANT SELECT ON PastEventsAttendance to secretary
GRANT SELECT ON BorrowersList to secretary
GRANT SELECT ON EventsThisMonth to secretary
GRANT SELECT ON ExamsStats to secretary
GRANT SELECT ON StudentsApprenticeship to secretary
GRANT SELECT ON Bilocations to secretary

GRANT EXECUTE ON GetProductName to secretary
GRANT EXECUTE ON GetUserIdFromUserEmail to secretary
GRANT EXECUTE ON GetParticipantIdFromUserAndProduct to secretary
GRANT SELECT ON ClientsExam to secretary
GRANT EXECUTE ON ClientsApprenticeships to secretary
GRANT EXECUTE ON CoursePass to secretary
GRANT SELECT ON CourseInfo to secretary
GRANT EXECUTE ON ModulesPresence to secretary
GRANT EXECUTE ON CoursesPresence to secretary
GRANT EXECUTE ON CoursesFreeSlots to secretary
GRANT SELECT ON ClientCourses to secretary
GRANT EXECUTE ON StudiesPass to secretary
GRANT EXECUTE ON StudiesPresence to secretary
GRANT EXECUTE ON CheckExamStatus to secretary
GRANT EXECUTE ON CheckExamDate to secretary
GRANT SELECT ON GetStudiesMeetings to secretary
GRANT SELECT ON GetRegisteredApprenticeship to secretary
GRANT EXECUTE ON checkApprenticeshipStatus to secretary
GRANT EXECUTE ON checkParicipantsLimit to secretary
GRANT EXECUTE ON checkIfStudiesMeetingParticipantsAllowed to secretary
GRANT SELECT ON GetStudiesMeetingAttendanceList to secretary
GRANT SELECT ON GetCourseModuleAttendanceList to secretary
GRANT EXECUTE ON checkIfClientPaid to secretary


GRANT SELECT on clientCourses to secretary
GRANT EXECUTE ON CheckIfCourseParticipantsAllowed to secretary
GRANT SELECT on getExamScores to secretary
GRANT EXECUTE on checkExamMaxPoints to secretary
GRANT EXECUTE on CheckApprenticeshipStatus to secretary
GRANT EXECUTE on checkParicipantsLimit to secretary



GRANT EXECUTE ON uspAddApprenticeship to secretary
GRANT EXECUTE ON uspAddUser to secretary
GRANT EXECUTE ON uspChangeMeetingDate to secretary
GRANT EXECUTE ON uspAddCourse to secretary
GRANT EXECUTE ON uspAddStudies to secretary
GRANT EXECUTE ON uspAddStudiesMeetings to secretary
GRANT EXECUTE ON uspAddWebinar to secretary
```

Manager

```
Create role manager

GRANT SELECT ON FinancialReport to manager
GRANT SELECT ON GraduationCandidates to manager
GRANT SELECT ON AllMeetings to manager
GRANT SELECT ON PastEventsAttendance to manager
GRANT SELECT ON BorrowersList to manager
GRANT SELECT ON EventsThisMonth to manager
GRANT SELECT ON ExamsStats to manager
GRANT SELECT ON StudentsApprenticeship to manager
GRANT SELECT ON Bilocations to manager

GRANT EXECUTE ON GetProductName to manager
GRANT EXECUTE ON GetUserIdFromUserEmail to manager
GRANT EXECUTE ON GetParticipantIdFromUserAndProduct to manager
GRANT SELECT ON ClientsExam to manager
GRANT EXECUTE ON ClientsApprenticeships to manager
GRANT EXECUTE ON CoursePass to manager
GRANT SELECT ON CourseInfo to manager
GRANT EXECUTE ON ModulesPresence to manager
GRANT EXECUTE ON CoursesPresence to manager
GRANT EXECUTE ON CoursesFreeSlots to manager
GRANT SELECT ON ClientCourses to manager
GRANT EXECUTE ON StudiesPass to manager
GRANT EXECUTE ON StudiesPresence to manager
GRANT EXECUTE ON CheckExamStatus to manager
GRANT EXECUTE ON CheckExamDate to manager
GRANT SELECT ON GetStudiesMeetings to manager
GRANT SELECT ON GetRegisteredApprenticeship to manager
GRANT EXECUTE ON checkApprenticeshipStatus to manager
GRANT EXECUTE ON checkParicipantsLimit to manager
GRANT EXECUTE ON checkIfStudiesMeetingParticipantsAllowed to manager
GRANT SELECT ON GetStudiesMeetingAttendanceList to manager
GRANT SELECT ON GetCourseModuleAttendanceList to manager
GRANT EXECUTE ON checkIfClientPaid to manager
GRANT EXECUTE ON CheckIfCourseParticipantsAllowed to manager
GRANT SELECT on getExamScores to manager

GRANT SELECT on clientCourses to manager
GRANT EXECUTE on checkExamMaxPoints to manager
GRANT EXECUTE on CheckApprenticeshipStatus to manager
GRANT EXECUTE on checkParicipantsLimit to manager
GRANT SELECT ON getPaymentHistory to manager


GRANT EXECUTE ON uspAddApprenticeship to manager
GRANT EXECUTE ON uspAddUser to manager
GRANT EXECUTE ON uspChangeMeetingDate to manager
GRANT EXECUTE ON uspAddCourse to manager
GRANT EXECUTE ON uspAddStudies to manager
GRANT EXECUTE ON uspAddStudiesMeetings to manager
GRANT EXECUTE ON uspAddWebinar to manager
GRANT EXECUTE ON uspSetCoursePrice to manager
GRANT EXECUTE ON uspSetMeetingPrice to manager
GRANT EXECUTE ON uspSetStudiesPrice to manager
GRANT EXECUTE ON uspSetWebinarPrice to manager
GRANT EXECUTE ON uspSetParticipantsLimit to manager
GRANT EXECUTE ON uspDeleteProduct to manager
```

Nauczyciel

```
Create role teacher

GRANT SELECT ON getTaughtWebinars to teacher
GRANT SELECT ON getTaughtCurses to teacher
GRANT SELECT ON getTaughtStudiesMeetings to teacher
GRANT SELECT ON getTaughtStudies to teacher
GRANT SELECT ON getStudiesMeetingAttendanceList to teacher
GRANT SELECT ON getCourseModuleAttendanceList to teacher
GRANT SELECT ON getTaughtWebinars to teacher

GRANT EXECUTE on uspAddExamResult to teacher
GRANT EXECUTE on uspAddMeetingPresence to teacher
GRANT EXECUTE on uspAddModulePresence to teacher
GRANT EXECUTE on uspSetMeetingPresence to teacher
```

Klient

```
Create role client

GRANT EXECUTE on CoursePass to client
GRANT SELECT on CourseInfo to client
GRANT EXECUTE on ModulesPresence to client
GRANT EXECUTE on CoursesPresence to client
GRANT EXECUTE on coursesFreeSlots to client
GRANT SELECT on ClientCourses to client
GRANT EXECUTE on StudiesPass to client
GRANT EXECUTE on StudiesPresence to client
GRANT SELECT on getExamScores to client
GRANT EXECUTE on checkExamStatus to client
GRANT EXECUTE on checkExamMaxPoints to client
GRANT EXECUTE on checkExamDate to client
GRANT SELECT on getStudiesmeetings to client
GRANT SELECT on getRegisteredApprenticeship to client
GRANT EXECUTE on CheckApprenticeshipStatus to client
GRANT EXECUTE on checkParicipantsLimit to client

GRANT SELECT on getOwnedWebinars to client
GRANT SELECT on getOwnedStudies to client
GRANT SELECT on getOwnedStudiesMeetings to client
GRANT SELECT on getOwnedCourses to client
GRANT SELECT on clientCourses to client
GRANT EXECUTE ON CheckIfCourseParticipantsAllowed to client
GRANT EXECUTE ON StudiesPass to client
GRANT SELECT ON getBucket to client
GRANT SELECT ON getPaymentHistory to client

GRANT EXECUTE on uspAddProductToOrder to client
GRANT EXECUTE on uspCancelPayment to client
GRANT EXECUTE on uspChangeToFullPrice to client
GRANT EXECUTE on uspDeleteProductFromOrder to client
GRANT EXECUTE on uspPay to client
GRANT EXECUTE on uspAddOrder to client
```

Właściciel

```
Create role owner

grant all privileges ON u_stankiew to owner
```

Indeksy

```
-- imię i nazwisko użytkownika
create index Users_last_name_index
    on Users (last_name)
go

-- adres użytkownika
create index Users_zip_code_index
    on Users (zip_code)
go

--typ produktu
create index Products_product_type_id_index
    on Products (product_type_id)
go

--język
create index Products_language_index
    on Products (language)
go

--numer zamówienia
create index Payments_order_id_index
    on Payments (order_id)
go

--data zamówienia
create index Payments_payment_date_index
    on Payments (payment_date)
go

--nazwa webinaru
create index Webinars_webinar_name_index
    on Webinars (webinar_name)
go

--data publikacji webinaru
create index Webinars_posted_date_index
    on Webinars (posted_date)
go

--nazwa kursu
create unique index Courses_course_name_uindex
    on Courses (course_name)
go

--data rozpoczęcia i zakończenia kursu
create unique index Courses_start_date_end_date_uindex
    on Courses (start_date, end_date)
go
--nazwa modułu
create unique index Uniq_Modules
    on Modules (module_name)
go

--id modułu
create index Modules_product_id_index
    on Modules (product_id)
```

```
go

--data rozpoczęcia i zakończenia modułu
create index Modules_start_date_index
    on Modules (start_date)
go

--sala, w której odbywa się moduł
create index Modules_classroom_index
    on Modules (classroom)
go

--nazwa studiów
create index Studies_name_index
    on Studies (name)
go

--id klienta, który jest uczestnikiem studiów
create index StudiesParticipants_client_id_index
    on StudiesParticipants (client_id)
go
--id studiów
create index StudiesParticipants_product_id_index
    on StudiesParticipants (product_id)
go

--id studiów
create index Exams_studies_id_index
    on Exams (studies_id)
go

--data egzaminu
create index Exams_date_index
    on Exams (date)
go

--data praktyk i id uczestnika studiów
create unique clustered index Apprenticeship_participant_id_date_uindex
    on Apprenticeship (participant_id, date)
go

--id studiów
create index StudiesMeetings_studies_id_index
    on StudiesMeetings (studies_id)
go
--data studiów
create index StudiesMeetings_date_index
    on StudiesMeetings (date)
go
```