

Projekt Bazy Danych na rok akademicki 0 0

Autorzy: Szymon Tyburczy oraz Marek Swakoń

Spis treści

Użytkownicy systemu

Klient
Administrator
Koordynator
Prowadzący
Tłumacz

Funkcje użytkowników

- **Klient:**
 - Przegląd studiów, kursów i webinarów
 - Składanie zamówienia
 - Uczestniczenie w przypisanych zajęciach
 - Sprawdzenie dostępnych miejsca na studiach i kursach
- **Prowadzący/Tłumacz:**
 - Dodawanie webinarów
 - Generowanie harmonogramu zajęć
 - Sprawdzenie listy obecności na zajęciach
 - Obliczanie frekwencji uczestników
- **Koordynator:**
 - Dodawanie nowych kursów
 - Dodawanie nowych studiów
 - Przypisywanie przedmiotów do studiów
 - Planowanie spotkań studiów
 - Monitorowanie przypisanych programów studiów
 - Generowanie harmonogramów studiów
 - Analiza dat praktyk
 - Aktualizacja danych kursów
 - Generowanie podsumowania frekwencji
 - Identyfikacja nieaktywnych użytkowników
 - Sprawdzanie ocen studentów
- **Administrator:**
 - Dodawanie nowych użytkowników
 - Usuwanie użytkowników
 - Usuwanie zamówień
 - Dodawanie pracowników
 - Dodawanie pozycji
 - Analiza przychodów z wydarzeń

Funkcje systemu

- **Zarządzanie użytkownikami:**

- Dodawanie usuwanie oraz edytowanie informacji o użytkowniku
- Obsługiwanie ról z różnymi dostępami
- Analizy aktywności użytkowników

- **Zarządzanie wydarzeniami:**

- Tworzenie i zarządzanie kursami studiami i webinarami
- Planowanie spotkań, przypisywanie tłumaczy i prowadzących
- kontrola dostępności miejsc oraz analizę frekwencji

- **Obsługa zamówień:**

- Obsługa składania zamówień na kursy, studia i webinar
- arządzanie szczegółami płatności, generowanie raportów finansowych oraz analizę wydatków klientów

- **Zarządzanie frekwencją i harmonogramami:**

- Monitorowanie frekwencji uczestników na wydarzeniach
- Generowanie raportów dotyczących obecności
- Identyfikacja potencjalnych konfliktów w harmonogramach prowadzących i uczestników

- **Obsługa praktyk:**

- arządzanie praktykami studenckimi
- Monitorowanie postępów i weryfikacja ukończenia zgodnie z wymaganiami

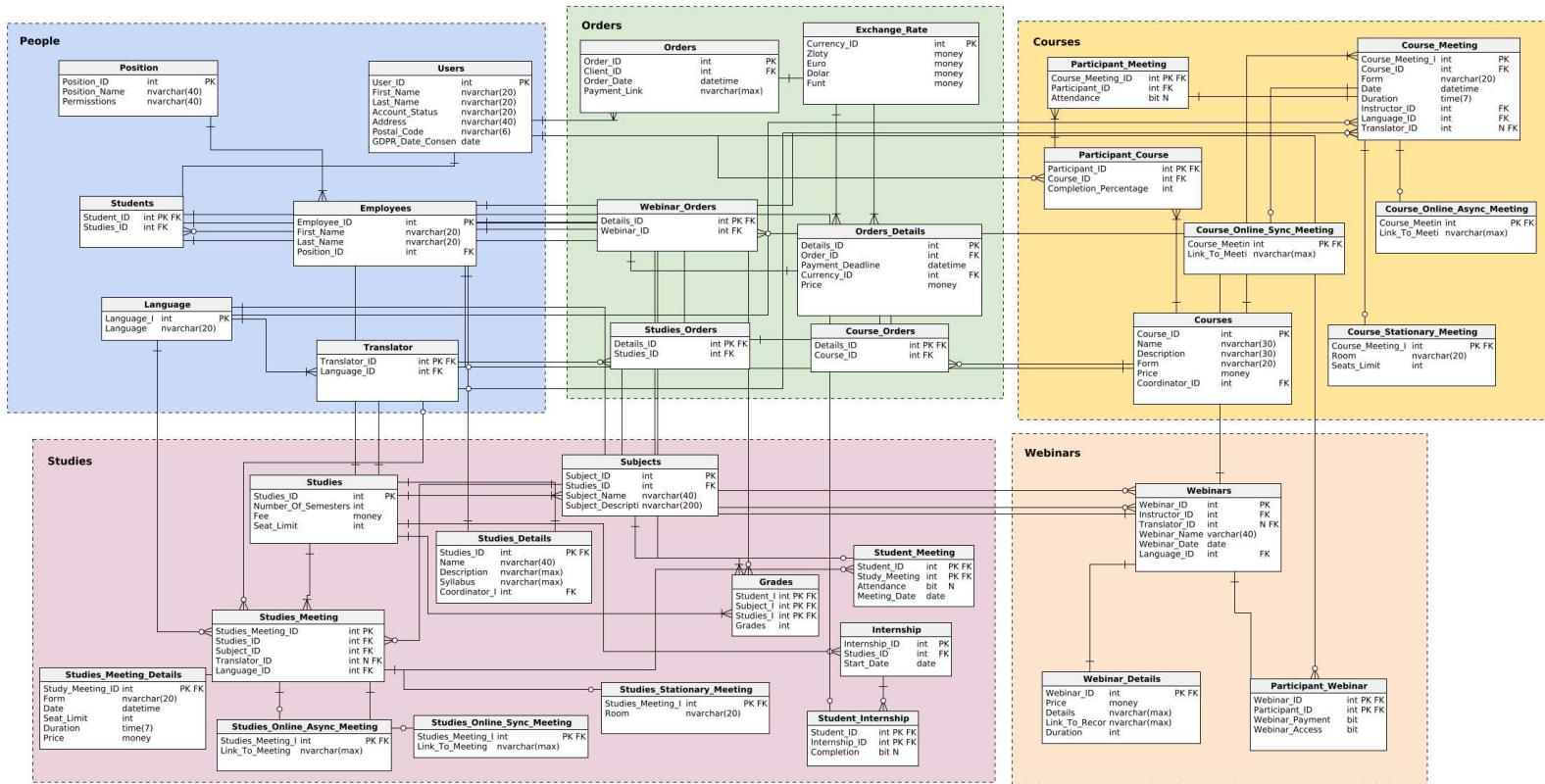
- **Analizy finansowe:**

- Raporty przychodów z podziałem na typy wydarzeń
- Analizę finansową na poziomie użytkowników i wydarzeń

- **Obsługa językowa:**

- Przypisywanie tłumaczy i języków do wydarzeń

Schemat



Opisy Tabel

Users

Tabela *Users* przechowuje dane wszystkich klientów. Zawiera informacje podstawowe, np. imię, nazwisko, status konta oraz dane adresowe.

Warunki integralnościowe:

- Każdy użytkownik musi mieć unikalny identyfikator *User_ID*
- Status konta (*Account_Status*) jest wymagany i może przyjmować wartości takie jak *open* lub *not open*
- Data zgody RODO (*GDPR_Date_Consent*) musi być podana

```
CREATE TABLE Users (
    User_ID          int            NOT NULL,
    First_Name       nvarchar(20)   NOT NULL,
    Last_Name        nvarchar(20)   NOT NULL,
    Account_Status   nvarchar(20)   NOT NULL,
    Address          nvarchar(40)   NOT NULL,
    Postal_Code      nvarchar(12)   NOT NULL,
    GDPR_Date_Consent date         NOT NULL,
    CONSTRAINT Users_pk PRIMARY KEY (User_ID)
);
```

Orders

Tabela *Orders* przechowuje informacje o zamówieniach składanych przez użytkowników. Warunki

integralnościowe:

- każde zamówienie musi być przypisane do istniejącego użytkownika (*Client_ID*)
- Pole *Order_Date* przechowuje datę zamówienia i nie może być puste

```
CREATE TABLE Orders (
    Order_ID      int            NOT NULL,
    Client_ID     int            NOT NULL,
    Order_Date    datetime       NOT NULL,
    Payment_Link  nvarchar(max) NOT NULL,
    CONSTRAINT Orders_pk PRIMARY KEY (Order_ID),
    CONSTRAINT Orders_Users_User_ID_fk FOREIGN KEY (Client_ID) REFERENCES Users
);
```

Orders_Details

Tabela Orders_Details przechowuje szczegóły zamówień, w tym kwotę, walutę i termin płatności.

Warunki integralnościowe:

- ażdy szczegół zamówienia musi być przypisany do zamówienia w tabeli Orders (Details_ID)
- Waluta (Currency_ID) musi istnieć w tabeli Exchange_Rate

```
CREATE TABLE Orders_Details (
    Details_ID      int      NOT NULL,
    Payment_Deadline datetime NOT NULL,
    Currency_ID     int      NOT NULL,
    Amount          int,
    CONSTRAINT Orders_Details_pk PRIMARY KEY (Details_ID),
    CONSTRAINT Orders_OrderDetails FOREIGN KEY (Details_ID) REFERENCES Orders,
    CONSTRAINT ExchangeRate_OrderDetails FOREIGN KEY (Currency_ID) REFERENCES Exchange_Rate
);
```

Course_Orders

Tabela Course_Orders przechowuje informacje o zamówieniach dotyczących kursów.

Warunki integralnościowe:

- ażdy rekord musi być przypisany do zamówienia z tabeli Orders_Details oraz kursu z tabeli Courses.

```
CREATE TABLE Course_Orders (
    Details_ID int NOT NULL,
    Course_ID  int NOT NULL,
    CONSTRAINT Course_Orders_pk PRIMARY KEY (Details_ID),
    CONSTRAINT OrderDetails_CourseOrders FOREIGN KEY (Details_ID) REFERENCES Orders_Details,
    CONSTRAINT Courses_CoursesOrders FOREIGN KEY (Course_ID) REFERENCES Courses
);
```

Studies_Orders

Tabela Studies_Orders przechowuje informacje o zamówieniach na studia.

Warunki integralnościowe:

- ażdy rekord musi być powiązany z tabelami Orders_Details i Studies.

```
CREATE TABLE Studies_Orders (
    Details_ID int NOT NULL,
    Studies_ID int NOT NULL,
    CONSTRAINT Studies_Orders_pk PRIMARY KEY (Details_ID),
    CONSTRAINT Studies_Orders_Orders_Details_Details_ID_fk FOREIGN KEY (Details_ID) REFERENCES Orders_Details,
    CONSTRAINT Studies_Orders_Studies_Studies_ID_fk FOREIGN KEY (Studies_ID) REFERENCES Studies
);
```

Webinar_Orders

Tabela Webinar_Orders przechowuje informacje o zamówieniach na webinary.

Warunki integralnościowe:

- ażdy rekord musi być powiązany z tabelami Orders_Details i Webinars.

```
CREATE TABLE Webinar_Orders (
    Details_ID int NOT NULL,
    Webinar_ID int NOT NULL,
    CONSTRAINT Webinar_Orders_pk PRIMARY KEY (Details_ID),
    CONSTRAINT Webinar_Orders_Orders_Details_Details_ID_fk FOREIGN KEY (Details_ID) REFERENCES Orders_Details,
    CONSTRAINT Webinar_Orders_Webinars_Webinar_ID_fk FOREIGN KEY (Webinar_ID) REFERENCES Webinars
);
```

Webinars

Tabela Webinars przechowuje informacje o webinarach, w tym język, prowadzącego oraz tłumacza.

Warunki integralnościowe:

- ażdy rekord musi być powiązany z tabelami Employees, Translator oraz Language.

```
CREATE TABLE Webinars (
    Webinar_ID      int      NOT NULL,
    Instructor_ID   int      NOT NULL,
    Translator_ID   int,
    Webinar_Name    nvarchar(80),
    Webinar_Date    datetime NOT NULL,
    Language_ID     int      NOT NULL,
    CONSTRAINT Webinars_pk PRIMARY KEY (Webinar_ID),
    CONSTRAINT Webinars_Employees_Employee_ID_fk FOREIGN KEY (Instructor_ID) REFERENCES Employees,
    CONSTRAINT Webinars_Translator_Translator_ID_fk FOREIGN KEY (Translator_ID) REFERENCES Translator,
    CONSTRAINT Language_Webinars FOREIGN KEY (Language_ID) REFERENCES Language
);
```

Webinar_Details

Tabela Webinar_Details przechowuje szczegóły dotyczące webinarów, takie jak cena, link do nagrania i czas trwania.

Warunki integralnościowe:

- ażdy rekord musi być powiązany z tabelą Webinars

```
CREATE TABLE Webinar_Details (
    Webinar_ID      int      NOT NULL,
    Price           money    NOT NULL,
    Link_To_Recording nvarchar(max) NOT NULL,
    Duration        int,
    CONSTRAINT Webinar_Details_pk PRIMARY KEY (Webinar_ID),
    CONSTRAINT Webinar_Details_Webinars_Webinar_ID_fk FOREIGN KEY (Webinar_ID) REFERENCES Webinars
);
```

Courses

Tabela Courses przechowuje informacje o kursach dostępnych w systemie, w tym ich nazwę, opis, cenę i koordynatora odpowiedzialnego za kurs.

Warunki integralnościowe:

- kurs musi mieć przypisanego koordynatora (wartość Coordinator_ID musi istnieć w tabeli Employees)

```
CREATE TABLE Courses (
    Course_ID INT NOT NULL,
    Name NVARCHAR(30) NOT NULL,
    Description NVARCHAR(30) NOT NULL,
    Price MONEY NOT NULL,
    Coordinator_ID INT NOT NULL,
    CONSTRAINT Courses_pk PRIMARY KEY (Course_ID),
    CONSTRAINT Courses_Employees_Employee_ID_fk FOREIGN KEY (Coordinator_ID) REFERENCES Employees(Employee_ID)
);
```

Course_Meeting

Tabela Course_Meeting zawiera informacje o spotkaniach organizowanych w ramach kursów, w tym ich formę, datę, język prowadzenia oraz tłumacza.

Warunki integralnościowe:

- Spotkanie musi być powiązane z istniejącym kursem (Course_ID)
- każde spotkanie wymaga przypisanego prowadzącego (Instructor_ID) oraz języka (Language_ID)

```
CREATE TABLE Course_Meeting (
    Course_Meeting_ID INT NOT NULL,
    Course_ID INT NOT NULL,
    Form NVARCHAR(20) NOT NULL,
    Date DATETIME NOT NULL,
    Instructor_ID INT NOT NULL,
    Language_ID INT NOT NULL,
    Translator_ID INT,
    CONSTRAINT Course_Meeting_pk PRIMARY KEY (Course_Meeting_ID),
    CONSTRAINT Courses_CourseMeeting FOREIGN KEY (Course_ID) REFERENCES Courses(Course_ID),
    CONSTRAINT Employees_CourseMeeting FOREIGN KEY (Instructor_ID) REFERENCES Employees(Employee_ID),
    CONSTRAINT Language_CourseMeeting FOREIGN KEY (Language_ID) REFERENCES Language(Language_ID),
    CONSTRAINT Course_Meeting_Translator_Translator_ID_fk
        FOREIGN KEY (Translator_ID) REFERENCES Translator(Translator_ID)
);
```

Participant_Course

Tabela Participant_Course przechowuje informacje o uczestnikach przypisanych do kursów oraz o ich postępach w realizacji kursu.

Warunki integralnościowe:

- ażdy uczestnik musi być przypisany do istniejącego kursu (Course_ID)

```
CREATE TABLE Participant_Course (
    Participant_ID INT NOT NULL,
    Course_ID INT NOT NULL,
    Completion_Percentage INT NOT NULL,
    CONSTRAINT Participant_Course_pk PRIMARY KEY (Participant_ID, Course_ID),
    CONSTRAINT Participant_Course_Users_User_ID_fk FOREIGN KEY (Participant_ID) REFERENCES Users(User_ID),
    CONSTRAINT Courses_ParticipantCourse FOREIGN KEY (Course_ID) REFERENCES Courses(Course_ID)
);
```

Participant_Meeting

Tabela Participant_Meeting przechowuje informacje o uczestnikach przypisanych do konkretnych spotkań kursowych oraz ich obecności.

Warunki integralnościowe:

- ażdy uczestnik musi być przypisany do istniejącego spotkania kursowego (Course_Meeting_ID)

```
CREATE TABLE Participant_Meeting (
    Participant_ID INT NOT NULL,
    Attendance BIT,
    Course_Meeting_ID INT NOT NULL,
    CONSTRAINT Participant_Meeting_pk PRIMARY KEY (Participant_ID, Course_Meeting_ID),
    CONSTRAINT Participant_Meeting_Participant_Course_Participant_ID_fk
        FOREIGN KEY (Participant_ID) REFERENCES Participant_Course(Participant_ID),
    CONSTRAINT Participant_Meeting_Course_Meeting_Course_Meeting_ID_fk
        FOREIGN KEY (Course_Meeting_ID) REFERENCES Course_Meeting(Course_Meeting_ID)
);
```

Course_Online_Async_Meeting

Tabela Course_Online_Async_Meeting przechowuje informacje o spotkaniach kursowych prowadzonych w formie asynchronicznej, np. z dostępem do materiałów wideo.

Warunki integralnościowe:

- Spotkanie musi być powiązane z istniejącym spotkaniem w tabeli Course_Meeting

```
CREATE TABLE Course_Online_Async_Meeting (
    Course_Meeting_ID INT NOT NULL,
    Link_To_Meeting NVARCHAR(MAX) NOT NULL,
    CONSTRAINT Course_Online_Async_Meeting_pk PRIMARY KEY (Course_Meeting_ID),
    CONSTRAINT CourseMeeting_CourseOnlineAsyncMeeting
        FOREIGN KEY (Course_Meeting_ID) REFERENCES Course_Meeting(Course_Meeting_ID)
);
```

Course_Online_Sync_Meeting

Tabela Course_Online_Sync_Meeting przechowuje informacje o spotkaniach kursowych prowadzonych w formie synchronicznej, np. za pomocą wideokonferencji.

Warunki integralnościowe:

- Spotkanie musi być powiązane z istniejącym spotkaniem w tabeli Course_Meeting

```
CREATE TABLE Course_Online_Async_Meeting (
    Course_Meeting_ID INT NOT NULL,
    Link_To_Meeting NVARCHAR(MAX) NOT NULL,
    CONSTRAINT Course_Online_Async_Meeting_pk PRIMARY KEY (Course_Meeting_ID),
    CONSTRAINT CourseMeeting_CourseOnlineAsyncMeeting
        FOREIGN KEY (Course_Meeting_ID) REFERENCES Course_Meeting(Course_Meeting_ID)
);
```

Course_Stationary_Meeting

Tabela Course_Stationary_Meeting przechowuje informacje o spotkaniach kursowych prowadzonych w formie stacjonarnej, w określonych salach z ograniczoną liczbą miejsc.

Warunki integralnościowe:

- Spotkanie musi być powiązane z istniejącym spotkaniem w tabeli Course_Meeting

```
CREATE TABLE Course_Online_Sync_Meeting (
    Course_Meeting_ID INT NOT NULL,
    Link_To_Meeting NVARCHAR(MAX) NOT NULL,
    CONSTRAINT Course_Online_Sync_Meeting_pk PRIMARY KEY (Course_Meeting_ID),
    CONSTRAINT CourseMeeting_CourseOnlineSyncMeeting
        FOREIGN KEY (Course_Meeting_ID) REFERENCES Course_Meeting(Course_Meeting_ID)
);
```

Participant_Webinar

Tabela Participant_Webinar przechowuje informacje o uczestnikach zapisanych na webinary, ich statusie płatności oraz dostępie do materiałów związanych z webinarami.

Warunki integralnościowe:

- ażdy uczestnik musi być przypisany do istniejącego webinaru (Webinar_ID)
- Uczestnik nie może być zapisany wielokrotnie na ten sam webinar

```
CREATE TABLE Course_Online_Sync_Meeting (
    Course_Meeting_ID INT NOT NULL,
    Link_To_Meeting NVARCHAR(MAX) NOT NULL,
    CONSTRAINT Course_Online_Sync_Meeting_pk PRIMARY KEY (Course_Meeting_ID),
    CONSTRAINT CourseMeeting_CourseOnlineSyncMeeting
        FOREIGN KEY (Course_Meeting_ID) REFERENCES Course_Meeting(Course_Meeting_ID)
);
```

Employees

Tabela Employees przechowuje dane dotyczące pracowników zatrudnionych w systemie, w tym ich dane osobowe i przypisaną pozycję.

Warunki integralnościowe:

- ażdy pracownik musi mieć przypisaną istniejącą pozycję (Position_ID)

```
CREATE TABLE Employees (
    Employee_ID INT NOT NULL,
    First_Name NVARCHAR(20) NOT NULL,
    Last_Name NVARCHAR(20) NOT NULL,
    Position_ID INT NOT NULL,
    CONSTRAINT Employees_pk PRIMARY KEY (Employee_ID),
    CONSTRAINT Positions_Employees
        FOREIGN KEY (Position_ID) REFERENCES Position(Position_ID)
);
```

Position

Tabela Position przechowuje dane dotyczące różnych stanowisk dostępnych w systemie oraz przypisanych do nich uprawnień.

Warunki integralnościowe:

- ażde stanowisko musi mieć zdefiniowaną nazwę i przypisane uprawnienia

```
CREATE TABLE Position (
    Position_ID INT NOT NULL,
    Position_Name NVARCHAR(40) NOT NULL,
    Permissions NVARCHAR(MAX) NOT NULL,
    CONSTRAINT Position_pk PRIMARY KEY (Position_ID)
);
```

Language

Tabela Language przechowuje informacje o językach, które są używane w systemie, na przykład w kursach, webinarach lub spotkaniach.

Warunki integralnościowe:

- ażdy język musi mieć zdefiniowaną nazwę

```
CREATE TABLE Language (
    Language_ID INT NOT NULL,
    Language NVARCHAR(20) NOT NULL,
    CONSTRAINT Language_pk PRIMARY KEY (Language_ID)
);
```

Translator

Tabela Translator przechowuje dane dotyczące tłumaczy i ich powiązanych języków.

Warunki integralnościowe:

- ażdy tłumacz musi być pracownikiem zdefiniowanym w tabeli Employees
- ażdy tłumacz musi mieć przypisany istniejący język (Language_ID).

```
CREATE TABLE Translator (
    Translator_ID INT NOT NULL,
    Language_ID INT NOT NULL,
    CONSTRAINT Translator_pk PRIMARY KEY (Translator_ID),
    CONSTRAINT Employees_Translator
        FOREIGN KEY (Translator_ID) REFERENCES Employees(Employee_ID),
    CONSTRAINT Language_Translator
        FOREIGN KEY (Language_ID) REFERENCES Language(Language_ID)
);
```

E change_Rate

Tabela E change_Rate przechowuje dane dotyczące kursów walut używanych w systemie, które mogą być wykorzystywane przy przetwarzaniu płatności.

Warunki integralnościowe:

- Wszystkie wartości kursów walut muszą być większe od zera

```
CREATE TABLE Exchange_Rate (
    Currency_ID INT NOT NULL,
    Zloty MONEY NOT NULL,
    Euro MONEY NOT NULL,
    Dolar MONEY NOT NULL,
    Funt MONEY NOT NULL,
    CONSTRAINT Exchange_Rate_pk PRIMARY KEY (Currency_ID)
);
```

Students

Tabela Students przechowuje informacje o studentach i ich przypisaniu do określonych studiów.

Warunki integralnościowe:

- ażdy student musi być użytkownikiem zdefiniowanym w tabeli Users
- ażdy student musi być przypisany do istniejących studiów

```
CREATE TABLE Students (
    Student_ID INT NOT NULL,
    Studies_ID INT NOT NULL,
    CONSTRAINT Students_pk PRIMARY KEY (Student_ID),
    CONSTRAINT Students_Users_User_ID_fk
        FOREIGN KEY (Student_ID) REFERENCES Users(User_ID),
    CONSTRAINT Students_Studies_Studies_ID_fk
        FOREIGN KEY (Studies_ID) REFERENCES Studies(Studies_ID)
);
```

Studies

Tabela Studies przechowuje informacje o dostępnych programach studiów.

Warunki integralnościowe:

- Liczba semestrów musi być większa od zera
- Limit miejsc musi być większy od zera

```
CREATE TABLE Studies (
    Studies_ID INT NOT NULL,
    Number_Of_Semesters INT NOT NULL,
    Fee MONEY NOT NULL,
    Seat_Limit INT NOT NULL,
    CONSTRAINT Studies_pk PRIMARY KEY (Studies_ID)
);
```

Studies_Details

Tabela Studies_Details przechowuje szczegółowe informacje o programach studiów, takie jak opis, syllabus i koordynator.

Warunki integralnościowe:

- ażdy program studiów musi mieć przypisanego koordynatora.

```
CREATE TABLE Studies_Details (
    Studies_ID INT NOT NULL,
    Name NVARCHAR(40) NOT NULL,
    Description NVARCHAR(MAX) NOT NULL,
    Syllabus NVARCHAR(MAX) NOT NULL,
    Coordinator_ID INT NOT NULL,
    CONSTRAINT Studies_Details_pk PRIMARY KEY (Studies_ID),
    CONSTRAINT Studies_StudiesDetails
        FOREIGN KEY (Studies_ID) REFERENCES Studies(Studies_ID),
    CONSTRAINT Studies_Details_Employees_Employee_ID_fk
        FOREIGN KEY (Coordinator_ID) REFERENCES Employees(Employee_ID)
);
```

Grades

Tabela Grades przechowuje informacje o ocenach studentów przypisanych do określonych przedmiotów w ramach studiów.

Warunki integralnościowe:

- Tabela posiada klucz główny obejmujący kolumny Student_ID, Subject_ID i Studies_ID, co zapobiega duplikacji ocen dla tego samego studenta, przedmiotu i studiów
- Oceny muszą być liczbami całkowitymi

```
CREATE TABLE Grades (
    Student_ID INT NOT NULL,
    Subject_ID INT NOT NULL,
    Studies_ID INT NOT NULL,
    Grades INT NOT NULL,
    CONSTRAINT Grades_pk PRIMARY KEY (Student_ID, Subject_ID, Studies_ID),
    CONSTRAINT Student_Grades
        FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID),
    CONSTRAINT Studies_Grades
        FOREIGN KEY (Studies_ID) REFERENCES Studies(Studies_ID)
);
```

Subjects

Tabela Subjects przechowuje informacje o przedmiotach, które są częścią programów studiów.

Warunki integralnościowe:

- ażdy przedmiot musi być przypisany do istniejącego programu studiów

```
CREATE TABLE Subjects (
    Subject_ID INT NOT NULL,
    Studies_ID INT NOT NULL,
    Subject_Name NVARCHAR(40) NOT NULL,
    CONSTRAINT Subjects_pk PRIMARY KEY (Subject_ID),
    CONSTRAINT Subjects_Studies_Studies_ID_fk
        FOREIGN KEY (Studies_ID) REFERENCES Studies(Studies_ID)
);
```

Internship

Tabela Internship przechowuje informacje o dostępnych praktykach powiązanych z programami studiów.

Warunki integralnościowe:

- każda praktyka musi być przypisana do istniejącego programu studiów

```
CREATE TABLE Internship (
    Internship_ID INT NOT NULL,
    Studies_ID INT NOT NULL,
    Start_Date DATE NOT NULL,
    CONSTRAINT Internship_pk PRIMARY KEY (Internship_ID),
    CONSTRAINT Studies_Internship
        FOREIGN KEY (Studies_ID) REFERENCES Studies(Studies_ID)
);
```

Student_Internship

Tabela Student_Internship przechowuje informacje o praktykach, w których uczestniczą studenci.

Warunki integralnościowe:

- Tabela posiada klucz główny obejmujący kolumny Student_ID i Internship_ID, co uniemożliwia duplikację zapisów dotyczących tego samego studenta i praktyki

```
CREATE TABLE Student_Internship (
    Student_ID INT NOT NULL,
    Internship_ID INT NOT NULL,
    Completion TINYINT,
    CONSTRAINT Student_Internship_pk PRIMARY KEY (Student_ID, Internship_ID),
    CONSTRAINT Student_StudentInternship
        FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID),
    CONSTRAINT Internship_StudentInternship
        FOREIGN KEY (Internship_ID) REFERENCES Internship(Internship_ID)
);
```

Studies_Meeting

Tabela Studies_Meeting przechowuje informacje o spotkaniach związanych z programami studiów.

Warunki integralnościowe:

- każde spotkanie musi być przypisane do istniejącego programu studiów i przedmiotu
- język oraz prowadzący muszą istnieć w odpowiednich tabelach

```
CREATE TABLE Studies_Meeting (
    Studies_Meeting_ID INT NOT NULL,
    Studies_ID INT NOT NULL,
    Subject_ID INT NOT NULL,
    Translator_ID INT NULL,
    Language_ID INT NOT NULL,
    Instructor_ID INT NOT NULL,
    CONSTRAINT Studies_Meeting_pk PRIMARY KEY (Studies_Meeting_ID),
    CONSTRAINT Studies_Meeting_Studies_Details_Studies_ID_fk
        FOREIGN KEY (Studies_ID) REFERENCES Studies_Details(Studies_ID),
    CONSTRAINT Studies_Meeting_Subjects_Subject_ID_fk
        FOREIGN KEY (Subject_ID) REFERENCES Subjects(Subject_ID),
    CONSTRAINT Studies_Meeting_Translator_Translator_ID_fk
        FOREIGN KEY (Translator_ID) REFERENCES Translator(Translator_ID),
    CONSTRAINT Language_StudiesMeeting
        FOREIGN KEY (Language_ID) REFERENCES Language(Language_ID),
    CONSTRAINT Studies_Meeting_Employees_Employee_ID_fk
        FOREIGN KEY (Instructor_ID) REFERENCES Employees(Employee_ID)
);
```

Studies_Meeting_Details

Tabela Studies_Meeting_Details zawiera szczegółowe informacje o spotkaniach, takie jak ich forma, data, cena czy czas trwania.

Warunki integralnościowe:

- ażde szczegóły muszą być powiązane z istniejącym spotkaniem w tabeli Studies_Meeting

```
CREATE TABLE Studies_Meeting_Details (
    Study_Meeting_ID INT NOT NULL,
    Form NVARCHAR(20) NOT NULL,
    Date DATETIME NOT NULL,
    Seat_Limit INT NOT NULL,
    Price MONEY NOT NULL,
    Duration INT NULL,
    CONSTRAINT Studies_Meeting_Details_pk PRIMARY KEY (Study_Meeting_ID),
    CONSTRAINT StudiesMeeting_StudiesMeetingDetails
        FOREIGN KEY (Study_Meeting_ID) REFERENCES Studies_Meeting(Studies_Meeting_ID)
);
```

Student_Meeting

Tabela Student_Meeting przechowuje informacje o uczestnictwie studentów w spotkaniach związanych z ich programami studiów.

Warunki integralnościowe:

- ażdy zapis musi odnosić się do istniejącego studenta i spotkania
- eden student może być przypisany tylko raz do danego spotkania

```
CREATE TABLE Student_Meeting (
    Student_ID INT NOT NULL,
    Study_Meeting_ID INT NOT NULL,
    Attendance BIT,
    Meeting_Date DATE NOT NULL,
    CONSTRAINT Student_Meeting_pk PRIMARY KEY (Student_ID, Study_Meeting_ID),
    CONSTRAINT Student_StudentMeeting
        FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID),
    CONSTRAINT StudiesMeeting_Student_Meeting
        FOREIGN KEY (Study_Meeting_ID) REFERENCES Studies_Meeting(Studies_Meeting_ID)
);
```

Studies_Online_Async_Meeting

Tabela Studies_Online_Async_Meeting przechowuje informacje o asynchronicznych spotkaniach studiów online. ażde spotkanie jest przypisane do konkretnego spotkania studiów oraz zawiera link do materiałów zajęć.

Warunki integralnościowe:

- ażde asynchroniczne spotkanie musi odnosić się do istniejącego spotkania studiów
- Tylko jedno asynchroniczne spotkanie może być przypisane do danego spotkania studiów

```
CREATE TABLE Studies_Online_Async_Meeting (
    Studies_Meeting_ID INT NOT NULL PRIMARY KEY
        REFERENCES Studies_Meeting(Studies_Meeting_ID),
    Link_To_Meeting NVARCHAR(MAX) NOT NULL
);
```

Studies_Online_Sync_Meeting

Tabela Studies_Online_Sync_Meeting zawiera dane o synchronizowanych spotkaniach studiów online. Spotkania te odbywają się w czasie rzeczywistym, a tabela przechowuje link umożliwiający dołączenie do spotkania.

Warunki integralnościowe:

- ażde synchronizowane spotkanie musi odnosić się do istniejącego spotkania studiów
- edno spotkanie studiów może być powiązane tylko z jednym spotkaniem synchronizowanym

```
CREATE TABLE Studies_Online_Sync_Meeting (
    Studies_Meeting_ID INT NOT NULL PRIMARY KEY
        REFERENCES Studies_Meeting(Studies_Meeting_ID),
    Link_To_Meeting NVARCHAR(MAX) NOT NULL
);
```

Studies_Stationary_Meeting

Tabela Studies_Stationary_Meeting przechowuje dane o stacjonarnych spotkaniach studiów. Przewiera informacje o miejscu i maksymalnej liczbie uczestników.

Warunki integralnościowe:

- każde stacjonarne spotkanie musi odnosić się do istniejącego spotkania studiów
- jeden rekord może być przypisany tylko do jednego spotkania studiów

```
CREATE TABLE Studies_Stationary_Meeting (
    Studies_Meeting_ID INT NOT NULL PRIMARY KEY
        REFERENCES Studies_Meeting(Studies_Meeting_ID),
    Room NVARCHAR(20) NOT NULL,
    Seats_Limit INT NOT NULL
);
```

Funkcje

CalculateAttendancePercentage

Funkcja CalculateAttendancePercentage przyjmuje dwa argumenty: @Student_ID (ID studenta) i @Subject_ID (ID przedmiotu). Funkcja zwraca procentową frekwencję studenta na spotkaniach związanych z danym przedmiotem.

```
CREATE FUNCTION CalculateAttendancePercentage
(
    @Student_ID INT,
    @Subject_ID INT
)
RETURNS DECIMAL(5, 2)
AS
BEGIN
    DECLARE @TotalMeetings INT;
    DECLARE @AttendedMeetings INT;
    DECLARE @AttendancePercentage DECIMAL(5, 2);

    SELECT @TotalMeetings = COUNT(*)
    FROM Studies_Meeting
    WHERE Subject_ID = @Subject_ID;

    SELECT @AttendedMeetings = COUNT(*)
    FROM Student_Meeting SM
    INNER JOIN Studies_Meeting STM
        1..n<->1: ON SM.Study_Meeting_ID = STM.Studies_Meeting_ID
    WHERE SM.Student_ID = @Student_ID
        AND STM.Subject_ID = @Subject_ID AND SM.Attendance = 1;

    IF @TotalMeetings > 0
        SET @AttendancePercentage = CAST(@AttendedMeetings AS DECIMAL(5, 2))
                                / CAST(@TotalMeetings AS DECIMAL(5, 2)) * 100;
    ELSE
        SET @AttendancePercentage = 0;

    RETURN @AttendancePercentage;
END;
```

CalculateMeetingDurationInMinutes

Funkcja CalculateMeetingDurationInMinutes oblicza czas trwania spotkania w minutach na podstawie argumentu @Duration, który jest typem TIME.

```
CREATE FUNCTION CalculateMeetingDurationInMinutes (@Duration TIME)
RETURNS INT
AS
BEGIN
    RETURN DATEDIFF(MINUTE, '00:00:00', @Duration);
END;
```

GetAverageCoursePrice

Funkcja GetAverageCoursePrice oblicza średnią cenę wszystkich dostępnych kursów w systemie. Nie przyjmuje żadnych argumentów i zwraca wartość typu MONEY.

```
CREATE FUNCTION [dbo].[GetAverageCoursePrice]()
RETURNS MONEY
AS
BEGIN
    DECLARE @AveragePrice MONEY;

    SELECT @AveragePrice = AVG(Price) FROM Courses;

    RETURN @AveragePrice;
END;
```

GetAvailableCourseSeats

Funkcja GetAvailableCourseSeats zwraca liczbę dostępnych miejsc na danym spotkaniu kursu, przyjmując argument @CourseMeetingID (ID spotkania kursu).

```
CREATE FUNCTION GetAvailableCourseSeats(@CourseMeetingID INT)
RETURNS INT
AS
BEGIN
    DECLARE @AvailableSeats INT;
    DECLARE @MeetingType NVARCHAR(20);

    SELECT @MeetingType = Form
    FROM Course_Meeting
    WHERE Course_Meeting_ID = @CourseMeetingID;

    IF @MeetingType IN ('Sync', 'Async')
    BEGIN
        RETURN NULL;
    END

    SELECT @AvailableSeats = Seats_Limit - ISNULL(COUNT(PC.Participant_ID), 0)
    FROM Course_Stationary_Meeting CSM
    JOIN Course_Meeting CM
        1<->1: ON CSM.Course_Meeting_ID = CM.Course_Meeting_ID
    LEFT JOIN Participant_Meeting PC
        1<->0..n: ON CM.Course_Meeting_ID = PC.Course_Meeting_ID
    WHERE CM.Course_Meeting_ID = @CourseMeetingID
    GROUP BY CSM.Seats_Limit;

    RETURN @AvailableSeats;
END;
```

GetAvailableSeats

Funkcja GetAvailableSeats oblicza liczbę dostępnych miejsc na danych studiach, przyjmując argument @StudiesID (ID studiów). Jeśli studia nie istnieją, funkcja zwraca wartość 1.

```
CREATE FUNCTION GetAvailableSeats(@StudiesID INT)
RETURNS INT
AS
BEGIN
    DECLARE @AvailableSeats INT;

    IF NOT EXISTS (SELECT 1 FROM Studies WHERE Studies_ID = @StudiesID)
    BEGIN
        RETURN -1;
    END

    SELECT @AvailableSeats = S.Seat_Limit - ISNULL(COUNT(St.Student_ID), 0)
    FROM Studies S
    LEFT JOIN Students St 1<->0..n: ON S.Studies_ID = St.Studies_ID
    WHERE S.Studies_ID = @StudiesID
    GROUP BY S.Seat_Limit;

    RETURN @AvailableSeats;
END;
```

GetCourseSchedule

Funkcja GetCourseSchedule zwraca szczegółowy harmonogram dla kursu o podanym @CourseID (ID kursu). Funkcja zwraca tabelę zawierającą informacje o spotkaniach związanych z kursem, takie jak: identyfikator spotkania, nazwa kursu, forma spotkania, data i czas trwania spotkania, prowadzący oraz język.

```
CREATE FUNCTION GetCourseSchedule (@CourseID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT
        CM.Course_Meeting_ID AS MeetingID,
        C.Name AS CourseName,
        CM.Form AS MeetingForm,
        CM.Date AS MeetingDate,
        CM.Duration AS MeetingDuration,
        CONCAT(E.First_Name, ' ', E.Last_Name) AS InstructorName,
        L.Language AS Language
    FROM Course_Meeting CM
    JOIN Courses C 1..n<->1: ON CM.Course_ID = C.Course_ID
    LEFT JOIN Employees E 1..n<->1: ON CM.Instructor_ID = E.Employee_ID
    LEFT JOIN Language L 1..n<->1: ON CM.Language_ID = L.Language_ID
    WHERE CM.Course_ID = @CourseID
)
```

GetParticipantAttendance

Funkcja GetParticipantAttendance zwraca liczbę spotkań, w których uczestnik @ParticipantID brał udział w ramach kursu o ID @CourseID. Wynikiem funkcji jest liczba całkowita określająca obecności.

```
CREATE FUNCTION [dbo].[GetParticipantAttendance]
(
    @ParticipantID INT,
    @CourseID INT
)
RETURNS INT
AS
BEGIN
    DECLARE @AttendanceCount INT;

    SELECT @AttendanceCount = COUNT(*)
    FROM Participant_Meeting PM
    INNER JOIN Course_Meeting CM
        1..n<->1: ON PM.Course_Meeting_ID = CM.Course_Meeting_ID
    WHERE PM.Participant_ID = @ParticipantID
        AND CM.Course_ID = @CourseID
        AND PM.Attendance = 1;

    RETURN @AttendanceCount;
END;
```

GetParticipantCount

Funkcja GetParticipantCount zwraca liczbę uczestników zapisanych na dany kurs o ID @CourseID. Wynik funkcji to liczba całkowita, która reprezentuje całkowitą liczbę uczestników przypisanych do kursu.

```
CREATE FUNCTION [dbo].[GetParticipantCount](@CourseID INT)
RETURNS INT
AS
BEGIN
    DECLARE @Count INT;

    SELECT @Count = COUNT(*)
    FROM Participant_Course
    WHERE Course_ID = @CourseID;

    RETURN @Count;
END;
```

GetStudentGrades

Funkcja GetStudentGrades zwraca szczegółową tabelę ocen studenta o ID @StudentID. Wynikiem jest tabela zawierająca informacje o przedmiotach, ocenach, studiach oraz dane osobowe studenta.

```
CREATE FUNCTION GetStudentGrades (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT
        SU.Subject_Name AS SubjectName,
        G.Grades AS Grade,
        SD.Name AS StudiesName,
        U.First_Name AS FirstName,
        U.Last_Name AS LastName
    FROM
        Grades G
    JOIN
        Students ST [1..n<->1] ON G.Student_ID = ST.Student_ID
    JOIN
        Users U [1<->1] ON ST.Student_ID = U.User_ID
    JOIN
        Studies S [1..n<->1] ON G.Studies_ID = S.Studies_ID
    JOIN
        Studies_Details SD [1<->1] ON S.Studies_ID = SD.Studies_ID
    JOIN
        Subjects SU ON G.Subject_ID = SU.Subject_ID
    WHERE
        G.Student_ID = @StudentID
)
```

GetStudentSchedule

Funkcja GetStudentSchedule zwraca harmonogram studenta o ID @StudentID. Wynikiem funkcji jest tabela zawierająca szczegóły spotkań, w których uczestniczy student, takie jak ID spotkania, nazwa studiów, ID przedmiotu, język, forma spotkania i data spotkania.

```
CREATE FUNCTION GetStudentSchedule (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT
        gss.MeetingID,
        gss.StudyName,
        gss.SubjectID,
        gss.Language,
        gss.MeetingForm,
        gss.MeetingDate
    FROM dbo.GetStudiesSchedule(
        @StudiesID (SELECT Studies_ID
        FROM Students WHERE Student_ID = @StudentID)
    ) gss
)
```

GetStudiesSchedule

Funkcja GetStudiesSchedule zwraca harmonogram zajęć dla studiów o ID @StudiesID. Wynikiem funkcji jest tabela zawierająca szczegóły spotkań, takie jak ID spotkania, nazwa studiów, ID przedmiotu, język, forma spotkania i data spotkania.

```
CREATE FUNCTION GetStudiesSchedule (@StudiesID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT
        SM.Studies_Meeting_ID AS MeetingID,
        SD.Name AS StudyName,
        SM.Subject_ID AS SubjectID,
        L.Language AS Language,
        SMD.Form AS MeetingForm,
        SMD.Date AS MeetingDate
    FROM Studies_Meeting SM
    JOIN Studies_Details SD
        1..n<->1: ON SM.Studies_ID = SD.Studies_ID
    LEFT JOIN Language L
        1..n<->1: ON SM.Language_ID = L.Language_ID
    JOIN Studies_Meeting_Details SMD
        1<->1: ON SM.Studies_Meeting_ID = SMD.Study_Meeting_ID
    WHERE SM.Studies_ID = @StudiesID
)
```

GetTotalOrderValue

Funkcja GetTotalOrderValue oblicza łączną wartość zamówienia na podstawie jego ID @OrderID. Wynikiem funkcji jest wartość pieniężna reprezentująca sumę wszystkich płatności związanych z danym zamówieniem.

```
CREATE FUNCTION GetTotalOrderValue (@OrderID INT)
RETURNS MONEY
AS
BEGIN
    DECLARE @TotalValue MONEY;

    SELECT @TotalValue = SUM(OD.Amount)
    FROM Orders_Details OD
    WHERE OD.Details_ID = @OrderID;

    RETURN ISNULL(@TotalValue, 0);
END;
```

HasStudentCompletedInternships

Funkcja HasStudentCompletedInternships sprawdza, czy student o ID @StudentID ukończył wymagane praktyki. Wynikiem funkcji jest wartość typu BIT (1 oznacza, że student spełnił wymagania, 0 – że nie). Funkcja weryfikuje, czy praktyki odbyły się co najmniej raz na pół roku oraz czy wszystkie praktyki zostały ukończone.

```
CREATE FUNCTION HasStudentCompletedInternships (@StudentID INT)
RETURNS BIT
AS
BEGIN
    DECLARE @IsValid BIT;

    IF NOT EXISTS (
        SELECT 1
        FROM Internship I
        JOIN Student_Internship SI
            [1<->1..n: ON I.Internship_ID = SI.Internship_ID]
        JOIN Students S [1..n<->1: ON SI.Student_ID = S.Student_ID]
        WHERE S.Student_ID = @StudentID
            AND SI.Completion <> 1
    ) AND EXISTS (
        SELECT 1
        FROM Internship I
        JOIN Student_Internship SI
            [1<->1..n: ON I.Internship_ID = SI.Internship_ID]
        JOIN Students S [1..n<->1: ON SI.Student_ID = S.Student_ID]
        WHERE S.Student_ID = @StudentID
        GROUP BY YEAR(I.Start_Date), CEILING(MONTH(I.Start_Date) / 6.0)
        HAVING COUNT(I.Internship_ID) >= 1
    )
    BEGIN
        SET @IsValid = 1;
    END
    ELSE
    BEGIN
        SET @IsValid = 0;
    END

    RETURN @IsValid;
END;
```

IsStudentEnrolledInMeeting

Funkcja IsStudentEnrolledInMeeting sprawdza, czy student o ID @StudentID jest przypisany do konkretnego spotkania o ID @MeetingID. Wynikiem funkcji jest wartość typu BIT (1 oznacza, że student jest zapisany, 0 – że nie).

```
CREATE FUNCTION IsStudentEnrolledInMeeting (@StudentID INT, @MeetingID INT)
RETURNS BIT
AS
BEGIN
    DECLARE @IsEnrolled BIT;

    IF EXISTS (
        SELECT 1
        FROM Students S
        JOIN Studies_Meeting SM ON S.Studies_ID = SM.Studies_ID
        WHERE S.Student_ID = @StudentID
            AND SM.Studies_Meeting_ID = @MeetingID
    )
    BEGIN
        SET @IsEnrolled = 1;
    END
    ELSE
    BEGIN
        SET @IsEnrolled = 0;
    END

    RETURN @IsEnrolled;
END;
```

Funkcje

AddCourse

Procedura AddCourse umożliwia dodanie nowego kursu do systemu, przypisując mu nazwę, opis, cenę oraz identyfikator koordynatora.

```
CREATE PROCEDURE AddCourse
    @Name NVARCHAR(30),
    @Description NVARCHAR(30),
    @Price MONEY,
    @Coordinator_ID INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @NewCourseID INT;

    SELECT @NewCourseID = ISNULL(MAX(Course_ID), 0) + 1 FROM Courses;

    INSERT INTO Courses (Course_ID, Name, Description, Price, Coordinator_ID)
    VALUES (Course_ID @NewCourseID, Name @Name, Description @Description, Price @Price, Coordinator_ID @Coordinator_ID);

    SELECT @NewCourseID AS NewCourseID;
END;
```

AddEmployee

Procedura AddEmployee umożliwia dodanie nowego pracownika do systemu, w tym jego ID, imienia, nazwiska oraz stanowiska.

```
CREATE PROCEDURE AddEmployee
    @EmployeeID INT,
    @FirstName NVARCHAR(20),
    @LastName NVARCHAR(20),
    @Position INT
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (
        SELECT 1
        FROM Employees
        WHERE Employee_ID = @EmployeeID
    )
    BEGIN
        RAISERROR('Employee with this ID already exists.', 16, 1);
    END;

    BEGIN TRY
        INSERT INTO Employees (Employee_ID, First_Name, Last_Name, Position_ID)
        VALUES ( @EmployeeID, @FirstName, @LastName, @Position);

        PRINT 'Employee added successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;
        SELECT
            @ErrorMessage = ERROR_MESSAGE(),
            @ErrorSeverity = ERROR_SEVERITY(),
            @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
END;
```

AddInternship

Procedura AddInternship umożliwia dodanie nowej praktyki, przypisując jej ID, powiązane studia oraz datę rozpoczęcia.

```
CREATE PROCEDURE AddInternship
    @InternshipID INT,
    @StudiesID INT,
    @StartDate DATE
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (
        SELECT 1
        FROM Internship
        WHERE Internship_ID = @InternshipID
    )
    BEGIN
        RAISERROR('Internship with this ID already exists.', 16, 1);
    END;

    BEGIN TRY
        INSERT INTO Internship (Internship_ID, Studies_ID, Start_Date)
        VALUES (@InternshipID, @StudiesID, @StartDate);

        PRINT 'Internship added successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;
        SELECT
            @ErrorMessage = ERROR_MESSAGE(),
            @ErrorSeverity = ERROR_SEVERITY(),
            @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
```

AddLanguage

Procedura AddLanguage umożliwia dodanie nowego języka do systemu.

```
CREATE PROCEDURE AddLanguage
    @Language NVARCHAR(20)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @NewLanguage_ID INT;

    SELECT @NewLanguage_ID = ISNULL(MAX(Language_ID), 0) + 1 FROM Language;

    INSERT INTO Language (Language_ID, Language)
    VALUES ( @NewLanguage_ID, @Language);

    SELECT @NewLanguage_ID AS NewLanguageID;
END;
```

AddOrder

Procedura AddOrder umożliwia dodanie nowego zamówienia do systemu. Obsługuje różne typy zamówień, takie jak studia, webinary oraz kursy, a także rejestruje szczegóły płatności.

```
CREATE PROCEDURE AddOrder
    @ClientID INT,
    @OrderDate DATETIME,
    @PaymentLink NVARCHAR(MAX),
    @StudiesID INT = NULL,
    @WebinarID INT = NULL,
    @CourseID INT = NULL,
    @PaymentDeadline DATETIME,
    @CurrencyID INT,
    @Price MONEY
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @OrderID INT;

    SELECT @OrderID = ISNULL(MAX(Order_ID), 0) + 1 FROM Orders;

    IF NOT EXISTS (SELECT 1 FROM Users WHERE User_ID = @ClientID)
    BEGIN
        RAISERROR('Klient o podanym ID nie istnieje.', 16, 1);
    END;

    INSERT INTO Orders (Order_ID, Client_ID, Order_Date, Payment_Link)
    VALUES (@OrderID, @ClientID, @OrderDate, @PaymentLink);

    IF @StudiesID IS NOT NULL
    BEGIN
        INSERT INTO Studies_Orders (Details_ID, Studies_ID)
        VALUES (@OrderID, @StudiesID);
    END
    ELSE IF @WebinarID IS NOT NULL
    BEGIN
        INSERT INTO Webinar_Orders (Details_ID, Webinar_ID)
        VALUES (@OrderID, @WebinarID);
    END
    ELSE IF @CourseID IS NOT NULL
    BEGIN
        INSERT INTO Course_Orders (Details_ID, Course_ID)
        VALUES (@OrderID, @CourseID);
    END
    ELSE
    BEGIN
        RAISERROR('Nie podano poprawnego typu zamówienia.', 16, 1);
        RETURN;
    END;

    INSERT INTO Orders_Details (Details_ID, Payment_Deadline, Currency_ID, Amount)
    VALUES (@OrderID, @PaymentDeadline, @CurrencyID, @Price);

    PRINT 'Zamówienie zostało pomyślnie dodane.';
END;
```

AddOrdersDetails

Procedura AddOrdersDetails umożliwia dodanie szczegółów płatności do zamówienia. Waliduje wprowadzone dane, takie jak identyfikator zamówienia, termin płatności, walutę oraz kwotę.

```
CREATE PROCEDURE AddOrdersDetails
    @Details_ID INT,
    @Payment_Deadline DATETIME,
    @Currency_ID INT,
    @Amount DECIMAL(18, 2)
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        IF @Details_ID IS NULL OR @Details_ID <= 0
        BEGIN
            THROW 50000, 'Details_ID must be a positive integer.', 1;
        END;

        IF @Payment_Deadline IS NULL
        BEGIN
            THROW 50001, 'Payment_Deadline cannot be NULL.', 1;
        END;

        IF @Currency_ID IS NULL OR @Currency_ID <= 0
        BEGIN
            THROW 50002, 'Currency_ID must be a positive integer.', 1;
        END;

        IF NOT EXISTS (SELECT 1 FROM Orders WHERE Order_ID = @Details_ID)
        BEGIN
            THROW 50004, 'Details_ID does not exist in the Orders table.', 1;
        END;

        INSERT INTO Orders_Details (Details_ID, Payment_Deadline, Currency_ID, Amount)
        VALUES (@Details_ID, @Payment_Deadline, @Currency_ID, @Amount);

        PRINT 'Payment detail added successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;
        SELECT
            @ErrorMessage = ERROR_MESSAGE(),
            @ErrorSeverity = ERROR_SEVERITY(),
            @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
END;
```

AddParticipantToCourse

Procedura AddParticipantToCourse umożliwia zapisanie uczestnika na wybrany kurs, sprawdzając, czy uczestnik nie jest już zapisany.

```
CREATE PROCEDURE AddParticipantToCourse
    @ParticipantID INT,
    @CourseID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Courses WHERE Course_ID = @CourseID)
    BEGIN
        RAISERROR('The course does not exist.', 16, 1);
    END;

    IF EXISTS (SELECT 1 FROM Participant_Course WHERE Participant_ID = @ParticipantID AND Course_ID = @CourseID)
    BEGIN
        RAISERROR('The participant is already enrolled in this course.', 16, 1);
    END;

    BEGIN TRY
        INSERT INTO Participant_Course (Participant_ID, Course_ID, Completion_Percentage)
        VALUES ( @ParticipantID, @CourseID, 0);

        PRINT 'Participant added successfully to the course.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;
        SELECT
            @ErrorMessage = ERROR_MESSAGE(),
            @ErrorSeverity = ERROR_SEVERITY(),
            @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
END;
```

AddPosition

Procedura AddPosition umożliwia dodanie nowego stanowiska do systemu, określając jego nazwę oraz uprawnienia.

```
CREATE PROCEDURE AddPosition
    @Position_Name NVARCHAR(40),
    @Permissions NVARCHAR(40)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @NewPosition_ID INT;

    SELECT @NewPosition_ID = ISNULL(MAX(Position_ID), 0) + 1 FROM Position;

    INSERT INTO Position (Position_ID, Position_Name, Permissions)
    VALUES (@Position_ID @NewPosition_ID, @Position_Name, @Permissions);

    SELECT @NewPosition_ID AS NewPositionID;
END;
```

AddStudent

Procedura AddStudent umożliwia dodanie nowego studenta do określonego programu studiów. Procedura sprawdza, czy podany program studiów istnieje oraz czy student nie jest już przypisany do tych studiów.

```
CREATE PROCEDURE AddStudent
    @Student_ID INT,
    @Studies_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Studies WHERE Studies_ID = @Studies_ID)
    BEGIN
        RAISERROR('Podane Studies_ID nie istnieje w tabeli Studies.', 16, 1);
    END;

    IF EXISTS (SELECT 1 FROM Students WHERE Student_ID = @Student_ID AND Studies_ID = @Studies_ID)
    BEGIN
        RAISERROR('Student jest już przypisany do tych studiów.', 16, 1);
    END;

    INSERT INTO Students (Student_ID, Studies_ID)
    VALUES (@Student_ID, @Studies_ID);

    PRINT 'Student został pomyślnie dodany do studiów.';
END;
```

AddStudies

Procedura AddStudies umożliwia dodanie nowego programu studiów do systemu wraz z jego szczegółami, takimi jak opis, syllabus, liczba semestrów, opłaty, limity miejsc oraz identyfikator koordynatora. Procedura sprawdza istnienie koordynatora w systemie.

```
CREATE PROCEDURE AddStudies
    @Name NVARCHAR(40),
    @Description NVARCHAR(MAX),
    @Syllabus NVARCHAR(MAX),
    @NumberOfSemesters INT,
    @Fee MONEY,
    @SeatLimit INT,
    @CoordinatorID INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @StudiesID INT;

    IF NOT EXISTS (SELECT 1 FROM Employees WHERE Employee_ID = @CoordinatorID)
    BEGIN
        RAISERROR('Koordynator o podanym ID nie istnieje.', 16, 1);
    END;

    BEGIN TRY
        BEGIN TRANSACTION;

        SELECT @StudiesID = ISNULL(MAX(Studies_ID), 0) + 1 FROM Studies;

        INSERT INTO Studies (Studies_ID, Number_of_Semesters, Fee, Seat_Limit)
        VALUES (@StudiesID, @NumberOfSemesters, @Fee, @SeatLimit);

        INSERT INTO Studies_Details (Studies_ID, Name, Description, Syllabus, Coordinator_ID)
        VALUES (@StudiesID, @Name, @Description, @Syllabus, @CoordinatorID);

        COMMIT TRANSACTION;
        PRINT 'Studia zostały pomyślnie dodane.';
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        RAISERROR('Wystąpił błąd podczas dodawania studiów.', 16, 1);
    END CATCH;
END;
```

AddSubjectToStudies

Procedura AddSubjectToStudies umożliwia dodanie nowego przedmiotu do istniejącego programu studiów. Procedura generuje nowy identyfikator przedmiotu i dodaje go do systemu.

```
CREATE PROCEDURE AddSubjectToStudies
    @StudiesID INT,
    @SubjectName NVARCHAR(40)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SubjectID INT;

    BEGIN TRY
        BEGIN TRANSACTION;

        SELECT @SubjectID = ISNULL(MAX(Subject_ID), 0) + 1 FROM Subjects;

        INSERT INTO Subjects (Subject_ID, Studies_ID, Subject_Name)
        VALUES ( @SubjectID, @StudiesID, @SubjectName);

        COMMIT TRANSACTION;
        PRINT 'Przedmiot został pomyślnie dodany do studiów.';
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        RAISERROR('Wystąpił błąd podczas dodawania przedmiotu.', 16, 1);
    END CATCH;
END;
```

AddTranslator

Procedura AddTranslator umożliwia dodanie nowego tłumacza do systemu, określając jego identyfikator oraz język, w którym się specjalizuje.

```
CREATE PROCEDURE AddTranslator
    @TranslatorID INT,
    @Language_ID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM Translator WHERE Translator_ID = @TranslatorID)
    BEGIN
        RAISERROR('Tłumacz o podanym ID już istnieje.', 16, 1);
    END;

    BEGIN TRY
        INSERT INTO Translator (Translator_ID, Language_ID)
        VALUES (@TranslatorID, @Language_ID);

        PRINT 'Tłumacz został pomyślnie dodany.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;
        SELECT
            @ErrorMessage = ERROR_MESSAGE(),
            @ErrorSeverity = ERROR_SEVERITY(),
            @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
END;
```

AddUser

Procedura AddUser umożliwia dodanie nowego użytkownika do systemu. Przyjmuje informacje o użytkowniku, takie jak imię, nazwisko, status konta, adres, kod pocztowy oraz datę zgody na przetwarzanie danych osobowych.

```
CREATE PROCEDURE AddUser
    @User_ID INT,
    @First_Name NVARCHAR(30),
    @Last_Name NVARCHAR(30),
    @Account_Status NVARCHAR(10),
    @Address NVARCHAR(100),
    @Postal_Code NVARCHAR(10),
    @GDPR_Date_Consent DATE
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM Users WHERE User_ID = @User_ID)
    BEGIN
        RAISERROR('Użytkownik o podanym User_ID już istnieje.', 16, 1);
    END;

    INSERT INTO Users (User_ID, First_Name, Last_Name, Account_Status, Address, Postal_Code, GDPR_Date_Consent)
    VALUES (@User_ID, @First_Name, @Last_Name,
            @Account_Status, @Address, @Postal_Code, @GDPR_Date_Consent);

    PRINT 'Użytkownik został pomyślnie dodany.';
END;
```

AddWebinar

Procedura AddWebinar umożliwia dodanie nowego webinaru do systemu. Przyjmuje parametry dotyczące webinaru, takie jak identyfikator, prowadzący, tłumacz, tytuł, data i język.

```
CREATE PROCEDURE AddWebinar
    @WebinarID INT,
    @InstructorID INT,
    @TranslatorID INT,
    @Title NVARCHAR(100),
    @Date DATE,
    @LanguageID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM Webinars WHERE Webinar_ID = @WebinarID)
    BEGIN
        RAISERROR('Webinar o podanym ID już istnieje.', 16, 1);
    END;

    BEGIN TRY
        INSERT INTO Webinars (Webinar_ID, Instructor_ID, Translator_ID,
                             Webinar_Name, Webinar_Date, Language_ID)
        VALUES (@WebinarID, @InstructorID,
                @TranslatorID, @Title, @Date, @LanguageID);
        PRINT 'Webinar został pomyślnie dodany.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;
        SELECT
            @ErrorMessage = ERROR_MESSAGE(),
            @ErrorSeverity = ERROR_SEVERITY(),
            @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
END;
```

Add_Studies_Meeting

Procedura Add_Studies_Meeting umożliwia dodanie nowego spotkania studiów. Przyjmuje parametry takie jak identyfikator spotkania, identyfikator studiów, przedmiotu, tłumacza oraz języka.

```
CREATE PROCEDURE Add_Studies_Meeting
    @Studies_Meeting_ID INT,
    @Studies_ID INT,
    @Subject_ID INT,
    @Translator_ID INT = NULL,
    @Language_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Studies WHERE Studies_ID = @Studies_ID)
    BEGIN
        RAISERROR('Podane Studies_ID nie istnieje w tabeli Studies.', 16, 1);
    END;

    IF NOT EXISTS (SELECT 1 FROM Subjects WHERE Subject_ID = @Subject_ID)
    BEGIN
        RAISERROR('Podane Subject_ID nie istnieje w tabeli Subjects.', 16, 1);
    END;

    IF @Translator_ID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Translator WHERE Translator_ID = @Translator_ID)
    BEGIN
        RAISERROR('Podane Translator_ID nie istnieje w tabeli Translator.', 16, 1);
        RETURN;
    END;

    INSERT INTO Studies_Meeting (Studies_Meeting_ID, Studies_ID,
                                Subject_ID, Translator_ID, Language_ID)
    VALUES (@Studies_Meeting_ID, @Studies_ID, @Subject_ID,
            @Translator_ID, @Language_ID);

    PRINT 'Nowe spotkanie studiów zostało pomyślnie dodane.';
END;
```

Add_Studies_Online_Async_Meeting

Procedura Add_Studies_Online_Async_Meeting umożliwia dodanie nowego asynchronicznego spotkania online dla studiów. Przyjmuje parametry takie jak identyfikator spotkania i link do materiałów.

```
CREATE PROCEDURE Add_Studies_Online_Async_Meeting
    @Studies_Meeting_ID INT,
    @Link_To_Meeting NVARCHAR(255)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Studies_Online_Async_Meeting WHERE Studies_Meeting_ID = @Studies_Meeting_ID)
    BEGIN
        RAISERROR('Spotkanie o podanym Studies_Meeting_ID już istnieje.', 16, 1);
    END;

    INSERT INTO Studies_Online_Async_Meeting (Studies_Meeting_ID, Link_To_Meeting)
    VALUES (@Studies_Meeting_ID, @Link_To_Meeting);

    PRINT 'Nowe asynchroniczne spotkanie online zostało pomyślnie dodane.';
END;
```

Add_Studies_Online_Sync_Meeting

Procedura Add_Studies_Online_Sync_Meeting umożliwia dodanie nowego synchronicznego spotkania online dla studiów. Spotkanie to jest połączone z istniejącym spotkaniem studiów i zawiera link do spotkania online.

```
CREATE PROCEDURE Add_Studies_Online_Sync_Meeting
    @Studies_Meeting_ID INT,
    @Link_To_Meeting NVARCHAR(255)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Studies_Online_Sync_Meeting WHERE Studies_Meeting_ID = @Studies_Meeting_ID)
    BEGIN
        RAISERROR('Spotkanie o podanym Studies_Meeting_ID już istnieje.', 16, 1);
    END;

    INSERT INTO Studies_Online_Sync_Meeting (Studies_Meeting_ID, Link_To_Meeting)
    VALUES (@Studies_Meeting_ID, @Link_To_Meeting);

    PRINT 'Nowe synchroniczne spotkanie online zostało pomyślnie dodane.';
END;
```

CheckStudentInternship

Procedura CheckStudentInternship umożliwia weryfikację, czy student ukończył praktyki na czas z wymaganą frekwencją 100%. Procedura analizuje unikalne okresy praktyk oraz obecności studenta na przypisanych spotkaniach.

```
CREATE PROCEDURE CheckStudentInternship
    @StudentID INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @UniqueSemestersCount INT;

    SELECT @UniqueSemestersCount = COUNT(DISTINCT CAST(YEAR(I.Start_Date) AS VARCHAR(4)) +
                                             '-' + CAST((MONTH(I.Start_Date) - 1) / 6 + 1 AS VARCHAR(1)))
    FROM Internship I
    INNER JOIN Student_Internship SI [1<->1..n] ON I.Internship_ID = SI.Internship_ID
    WHERE SI.Student_ID = @StudentID;

    IF @UniqueSemestersCount = 0
    BEGIN
        PRINT 'Student nie ukończył żadnych praktyk.';
        RETURN;
    END;
    DECLARE @MissedMeetingsCount INT;
    SELECT @MissedMeetingsCount = COUNT(*)
    FROM Student_Meeting SM
    INNER JOIN Studies_Meeting STM [1..n<->1..1] ON SM.Study_Meeting_ID = STM.Studies_Meeting_ID
    INNER JOIN Internship I ON STM.Studies_ID = I.Studies_ID
    WHERE SM.Student_ID = @StudentID
        AND SM.Attendance = 0;
    IF @MissedMeetingsCount > 0
    BEGIN
        PRINT 'Student nie ma 100% obecności na praktykach.';
        RETURN;
    END;
    PRINT 'Student ukończył praktyki z wymaganą frekwencją.';

END;
```

DeleteInternship

Procedura DeleteInternship pozwala na usunięcie praktyki z systemu na podstawie jej identyfikatora. Przed usunięciem weryfikuje, czy podana praktyka istnieje w bazie danych.

```
CREATE PROCEDURE DeleteInternship
    @Internship_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Internship WHERE Internship_ID = @Internship_ID)
    BEGIN
        RAISERROR('Podane Internship_ID nie istnieje w tabeli Internships.', 16, 1);
    END;

    DELETE FROM Internship
    WHERE Internship_ID = @Internship_ID;

    PRINT 'Praktyka została pomyślnie usunieta.';
END;
```

DeleteOrder

Procedura DeleteOrder umożliwia usunięcie zamówienia z systemu, wraz z jego powiązanymi szczegółami, takimi jak szczegóły płatności, zamówione kursy, webinary czy studia.

```
CREATE PROCEDURE DeleteOrder
    @OrderID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Orders WHERE Order_ID = @OrderID)
    BEGIN
        RAISERROR('Zamówienie o podanym ID nie istnieje.', 16, 1);
    END;

    BEGIN TRY
        BEGIN TRANSACTION;

        DELETE FROM Studies_Orders WHERE Details_ID = @OrderID;
        DELETE FROM Webinar_Orders WHERE Details_ID = @OrderID;
        DELETE FROM Course_Orders WHERE Details_ID = @OrderID;
        DELETE FROM Orders_Details WHERE Details_ID = @OrderID;

        DELETE FROM Orders WHERE Order_ID = @OrderID;

        COMMIT TRANSACTION;
        PRINT 'Zamówienie zostało pomyślnie usunięte.';
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        RAISERROR('Wystąpił błąd podczas usuwania zamówienia.', 16, 1);
    END CATCH;
END;
```

DeleteUser

Procedura DeleteUser umożliwia usunięcie użytkownika z systemu na podstawie jego identyfikatora. Weryfikuje istnienie użytkownika w bazie przed wykonaniem operacji usunięcia.

```
CREATE PROCEDURE DeleteUser
    @User_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Users WHERE User_ID = @User_ID)
    BEGIN
        RAISERROR('Użytkownik o podanym User_ID nie istnieje.', 16, 1);
    END;

    DELETE FROM Users
    WHERE User_ID = @User_ID;

    PRINT 'Użytkownik został pomyślnie usunięty.';
END;
```

Delete_Studies_Meeting

Procedura Delete_Studies_Meeting pozwala na usunięcie spotkania studiów na podstawie jego identyfikatora. Przed usunięciem weryfikuje, czy dane spotkanie istnieje.

```
CREATE PROCEDURE Delete_Studies_Meeting
    @Studies_Meeting_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Studies_Meeting WHERE Studies_Meeting_ID = @Studies_Meeting_ID)
    BEGIN
        RAISERROR('Spotkanie o podanym Studies_Meeting_ID nie istnieje.', 16, 1);
    END;

    DELETE FROM Studies_Meeting
    WHERE Studies_Meeting_ID = @Studies_Meeting_ID;

    PRINT 'Spotkanie zostało pomyślnie usunięte.';
END;
```

Delete_Studies_Online_Async_Meeting

Procedura Delete_Studies_Online_Async_Meeting umożliwia usunięcie asynchronicznego spotkania online dla studiów. Weryfikuje istnienie spotkania przed jego usunięciem.

```
CREATE PROCEDURE Delete_Studies_Online_Async_Meeting
    @Studies_Meeting_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Studies_Online_Async_Meeting WHERE Studies_Meeting_ID = @Studies_Meeting_ID)
    BEGIN
        RAISERROR('Spotkanie o podanym Studies_Meeting_ID nie istnieje.', 16, 1);
    END;

    DELETE FROM Studies_Online_Async_Meeting
    WHERE Studies_Meeting_ID = @Studies_Meeting_ID;

    PRINT 'Spotkanie asynchroniczne online zostało pomyślnie usunięte.';
END;
```

Delete_Studies_Online_Sync_Meeting

Procedura Delete_Studies_Online_Sync_Meeting pozwala na usunięcie synchronicznego spotkania online dla studiów. Weryfikuje istnienie spotkania przed wykonaniem operacji usunięcia.

```
CREATE PROCEDURE Delete_Studies_Online_Sync_Meeting
    @Studies_Meeting_ID INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Studies_Online_Sync_Meeting WHERE Studies_Meeting_ID = @Studies_Meeting_ID)
    BEGIN
        RAISERROR('Spotkanie o podanym Studies_Meeting_ID nie istnieje.', 16, 1);
    END;

    DELETE FROM Studies_Online_Sync_Meeting
    WHERE Studies_Meeting_ID = @Studies_Meeting_ID;

    PRINT 'Spotkanie synchroniczne online zostało pomyślnie usunięte.';
END;
```

GetAttendanceSummary

Procedura GetAttendanceSummary generuje podsumowanie frekwencji uczestników dla różnych typów wydarzeń, takich jak webinar, kursy i spotkania studiów.

```
CREATE PROCEDURE GetAttendanceSummary
AS
BEGIN
    SET NOCOUNT ON;

    CREATE TABLE #AttendanceSummary (
        EventType NVARCHAR(20),
        EventName NVARCHAR(80),
        TotalParticipants INT,
        AttendedParticipants INT,
        AttendancePercentage DECIMAL(5, 2)
    );

    INSERT INTO #AttendanceSummary (EventType, EventName, TotalParticipants, AttendedParticipants, AttendancePercentage)
    SELECT EventType 'Webinar', EventName W.Webinar_Name, TotalParticipants COUNT(PW.Participant_ID),
           AttendedParticipants COUNT(CASE WHEN PW.Webinar_Access = 1 THEN 1 END),
           CAST((100.0 * COUNT(CASE WHEN PW.Webinar_Access = 1 THEN 1 END) / NULLIF(COUNT(PW.Participant_ID), 0)) AS DECIMAL(5, 2))
    FROM Webinars W
    LEFT JOIN Participant_Webinar PW 1<->0..n: ON W.Webinar_ID = PW.Webinar_ID
    GROUP BY W.Webinar_Name;

    INSERT INTO #AttendanceSummary (EventType, EventName, TotalParticipants, AttendedParticipants, AttendancePercentage)
    SELECT EventType 'Course', EventName C.Name, TotalParticipants COUNT(PC.Participant_ID),
           AttendedParticipants COUNT(CASE WHEN PC.Completion_Percentage >= 80 THEN 1 END),
           CAST((100.0 * COUNT(CASE WHEN PC.Completion_Percentage >= 80 THEN 1 END) / NULLIF(COUNT(PC.Participant_ID), 0)) AS DECIMAL(5, 2))
    FROM Courses C
    LEFT JOIN Participant_Course PC 1<->0..n: ON C.Course_ID = PC.Course_ID
    GROUP BY C.Name;

    INSERT INTO #AttendanceSummary (EventType, EventName, TotalParticipants, AttendedParticipants, AttendancePercentage)
    SELECT EventType 'Studies', CAST(SM.Study_Meeting_ID AS NVARCHAR(80)), TotalParticipants COUNT(SM.Student_ID),
           AttendedParticipants COUNT(CASE WHEN SM.Attendance = 1 THEN 1 END),
           CAST((100.0 * COUNT(CASE WHEN SM.Attendance = 1 THEN 1 END) / NULLIF(COUNT(SM.Student_ID), 0)) AS DECIMAL(5, 2))
    FROM Student_Meeting SM
    JOIN Studies_Meeting SDM 1..n<->1: ON SM.Study_Meeting_ID = SDM.Studies_Meeting_ID
    GROUP BY SM.Study_Meeting_ID;

    SELECT * FROM #AttendanceSummary;
```

GetFinancialReport

Procedura GetFinancialReport generuje raport finansowy, przedstawiający przychody z różnych typów wydarzeń: webinarów, kursów i studiów.

```
CREATE PROCEDURE GetFinancialReport
AS
BEGIN
    SET NOCOUNT ON;

    CREATE TABLE #FinancialReport (
        Category NVARCHAR(20),
        TotalRevenue MONEY
    );

    INSERT INTO #FinancialReport (Category, TotalRevenue)
    SELECT Category 'Webinars', TotalRevenue SUM(OD.Amount)
    FROM Orders_Details OD
    JOIN Webinar_Orders WO 1<->1: ON OD.Details_ID = WO.Details_ID;

    INSERT INTO #FinancialReport (Category, TotalRevenue)
    SELECT Category 'Courses', TotalRevenue SUM(OD.Amount)
    FROM Orders_Details OD
    JOIN Course_Orders CO 1<->1: ON OD.Details_ID = CO.Details_ID;

    INSERT INTO #FinancialReport (Category, TotalRevenue)
    SELECT Category 'Studies', TotalRevenue SUM(OD.Amount)
    FROM Orders_Details OD
    JOIN Studies_Orders SO 1<->1: ON OD.Details_ID = SO.Details_ID;

    SELECT Category, ISNULL(TotalRevenue, 0) AS TotalRevenue
    FROM #FinancialReport;

    DROP TABLE #FinancialReport;
END;
```

UpdateMeetingDuration

Procedura UpdateMeetingDuration służy do zmiany długości trwania spotkania w tabeli Course_Meeting.

```
CREATE PROCEDURE UpdateMeetingDuration
    @MeetingID INT,
    @NewDuration TIME
AS
BEGIN
    UPDATE Course_Meeting
    SET Duration = @NewDuration
    WHERE Course_Meeting_ID = @MeetingID;
END;
```