Podstawy baz danych Projekt

Bernard Gawor, Olgierd Smyka, Stas Kochevenko

1. **Funkcjonalność systemu**

Użytkownicy

Klient bez konta:

- Wyświetlanie oferowanych usług
- Utworzenie konta indywidualnego bądź firmowego

Klient indywidualny:

- Wyświetlanie oferowanych usług
- Możliwość zapisu na szkolenia bezpłatne lub płatne poprzez uiszczenie opłaty
- Wyświetlanie wszystkich kursów i szkoleń, na które jest zapisany oraz szczegółów ich dotyczących
- Dostęp do materiałów dydaktycznych w kursach i na studiach

Klient firmowy:

- Wyświetlanie oferowanych usług
- Możliwość zapisu na szkolenia bezpłatne lub płatne poprzez uiszczenie opłaty (indywidualnie bądź na firmę)
- Wyświetlanie wszystkich kursów i szkoleń, na które jest zapisany oraz szczegółów ich dotyczących
- Dostęp do materiałów dydaktycznych w kursach i na studiach.
- Wyświetlanie faktur za szkolenia

Pracownicy:

- Wyświetlanie wszystkich kursów i szkoleń, które prowadzi oraz szczegółów ich dotyczących oraz możliwość ich edycji
- Dostęp do materiałów dydaktycznych w kursach i na studiach oraz możliwość ich edycji

Administrator (dyrektor szkoły):

- Aktualizacja oferowanych usług
- Wystawianie faktur
- Edycja listy pracowników
- Zmiana uprawnień użytkownika
- Dostęp do funkcji systemowych oraz możliwość ich udostępnienia

System

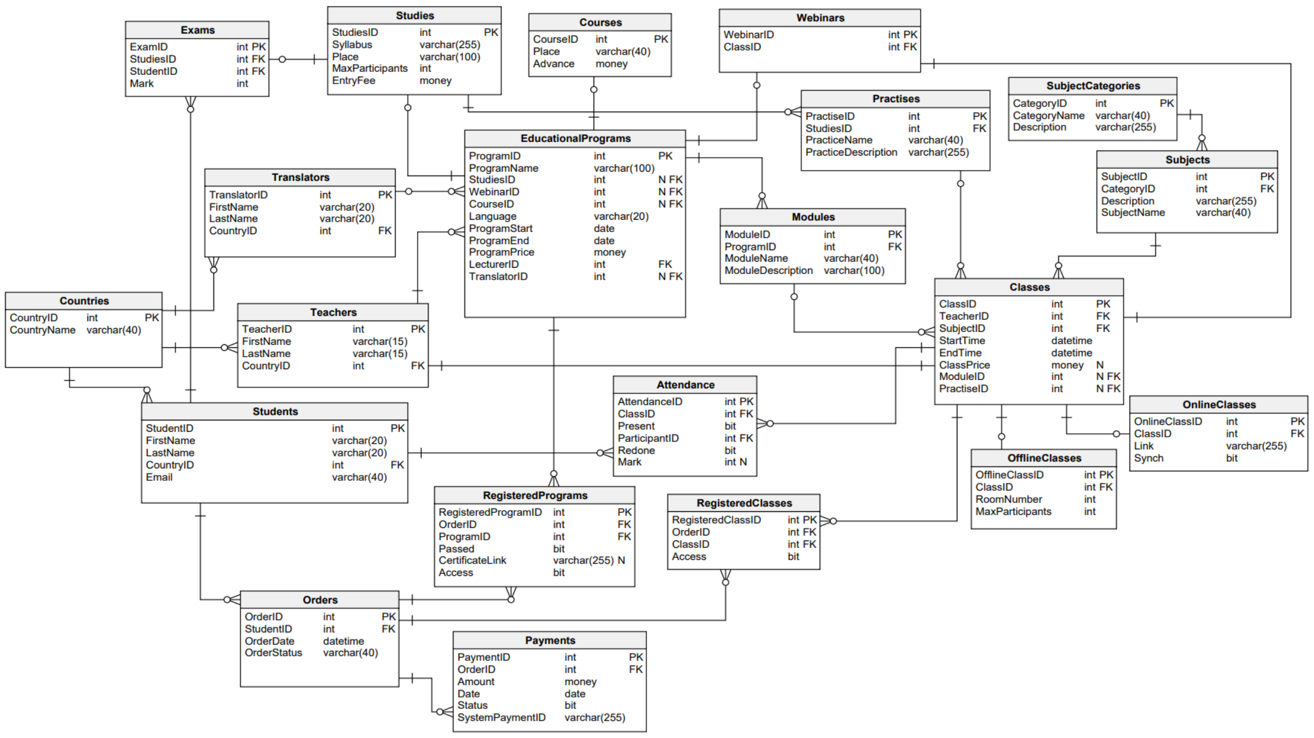
- Raporty finansowe – zestawienie przychodów dla każdego webinaru/kursu/studium
- Lista „dłużników” – osoby, które skorzystały z usług, ale nie uiściły opłat.
- Ogólny raport dotyczący liczby zapisanych osób na przyszłe wydarzenia (z informacją, czy wydarzenie jest stacjonarnie, czy zdalnie)

- Ogólny raport dotyczący frekwencji na zakończonych już wydarzeniach
- Lista obecności dla każdego szkolenia z datą, imieniem, nazwiskiem i informacją czy uczestnik był obecny, czy nie
- Raport bilokacji: lista osób, które są zapisane na co najmniej dwa przyszłe szkolenia, które ze sobą kolidują czasowo

Specyfikacje

- Kursy i studia mogą odbywać się: online, stacjonarnie, hybrydowo
- Stacjonarne zajęcia kursów i studiów posiadają limit miejsc
- Webinary udostępniane są uczestnikom na okres 30 dni
- Zaliczenie kursu wymaga zaliczenia min. 80% modułów
- W przypadku studiów wymagane jest zaliczenie praktyk oraz frekwencja na poziomie minimum 80%, przy czym nieobecności mogą zostać odrobione poprzez uczestnictwo w zajęciach lub kursie komercyjnym o zbliżonej tematyce
- Tematyka programów studiów nie może być modyfikowana po ich rozpoczęciu
- Praktyki trwają 14 dni – wymagana jest tu 100% frekwencja
- Możliwość zapisania się na pojedyncze spotkania bez konieczności udziału w całym studium, przy tym cena jest inna
- Administrator ma możliwość zapisu klientów na nieopłacone szkolenia
- Uczestnictwo w kursie wymaga wpłacenia zaliczki przy zapisie, oraz dopłaty całości kwoty najpóźniej 3 dni przed rozpoczęciem kursu
- Uczestnictwo w studium wymaga uiszczenia wpisowego oraz uiszczenia opłaty za dany zjazd najpóźniej 3 dni przed jego rozpoczęciem
- Szkolenia mogą być prowadzone w różnych ustalonych językach
- Wszystkie zajęcia online odbywają się na zewnętrznej platformie chmurowej
- System płatności jest dostarczany przez zewnętrzną firmę

2.Schemat Bazy Danych



Oferowane przez firmę usługi (różnego rodzaju kursy i szkolenia) łączy EducationalPrograms. Każdy rekord przedstawia albo studia (Studies), albo kurs (Courses) albo webinar (Webinars). Spis wszystkich poszczególnych zajęć (spotkań) znajduje się w tabeli Classes. Spotkania mogą być stacjonarne (OfflineClasses) lub niestacjonarne (OnlineClasses). Kursy (Courses) składają się z modułów (Modules). Pojedyncze zajęcia tych modułów mogą być prowadzone stacjonarnie lub niestacjonarnie. Studia podobnie do kursów składają się z modułów (Modules), oraz posiadają praktyki (Practises).

Studenci mogą składać zamówienia (Orders) i przeglądać listę programów (RegisteredPrograms) oraz pojedynczych spotkań (RegisteredClasses), na które są zapisane.

```sql
-- Table:  Translators
CREATE TABLE  Translators (
   TranslatorID int  NOT NULL,
   FirstName varchar(20)  NOT NULL,
   LastName varchar(20)  NOT NULL,
   CountryID int  NOT NULL,
   CONSTRAINT Translator_pk PRIMARY KEY  (TranslatorID)
);
```

```
-- Table: Attendance
-- Zawiera informacje dotyczące obecności konkretnych studentów z tabeli Students
na zajęciach z tabeli Classes
CREATE TABLE Attendance (
   AttendanceID int  NOT NULL,
   ClassID int  NOT NULL,
   Present bit  NOT NULL DEFAULT 0,
   ParticipantID int  NOT NULL,
   Redone bit  NOT NULL DEFAULT 0,
   CONSTRAINT Attendance_pk PRIMARY KEY  (AttendanceID)
);


-- Table: Classes
-- Pojedyncze spotkanie w ramach programu edukacyjnego (albo konkretnego modułu w
przypadku kursów lub studiów), może być w formacie online lub offline
CREATE TABLE Classes (
   ClassID int  NOT NULL,
   TeacherID int  NOT NULL,
   SubjectID int  NOT NULL,
   StartTime datetime  NOT NULL,
   EndTime datetime  NOT NULL,
   ClassPrice money  NULL,
   ModuleID int  NULL,
   PractiseID int  NULL,
   CHECK (EndTime > StartTime),
   CHECK (ClassPrice >= 0),
   CONSTRAINT Classes_pk PRIMARY KEY  (ClassID)
);


-- Table: Countries
CREATE TABLE Countries (
   CountryID int  NOT NULL,
   CountryName int  NOT NULL UNIQUE,
   CONSTRAINT Countries_pk PRIMARY KEY  (CountryID)
);


-- Table: Courses
CREATE TABLE Courses (
   CourseID int  NOT NULL,
   Place varchar(40)  NOT NULL,
   Advance money  NOT NULL,
   CHECK (Advance >= 0),
   CONSTRAINT Courses_pk PRIMARY KEY  (CourseID)
);
```

```
-- Table: EducationalPrograms
-- Zawiera szczegóły konkretnego programu edukacyjnego, którym mogą być studia z
tabeli Studies, kursy z tabeli Courses lub Webinary z tabeli Webinars, w każdym
rekorcie tylko jedna z trzech wartości: StudiesID, WebinarID, CourseID nie jest
NULL-em.
CREATE TABLE EducationalPrograms (
   ProgramID int  NOT NULL,
   ProgramName varchar(100)  NOT NULL UNIQUE,
   StudiesID int  NULL,
   WebinarID int  NULL,
   CourseID int  NULL,
   Language varchar(20)  NOT NULL,
   ProgramStart date  NOT NULL,
   ProgramEnd date  NOT NULL,
   ProgramPrice money  NOT NULL,
   LecturerID int  NOT NULL DEFAULT ‘Polish’,
   TranslatorID int  NULL,
   CHECK (ProgramEnd > ProgramStart),
   CHECK (ProgramPrice >= 0),
   CONSTRAINT EducationalPrograms_pk PRIMARY KEY  (ProgramID)
);


-- Table: Exams
-- Zawiera wyniki z egzaminów dla studentów (Tabela Students) zapisanych na
studia(Tabela Studies)
CREATE TABLE Exams (
   ExamID int  NOT NULL,
   StudiesID int  NOT NULL,
   StudentID int  NOT NULL,
   Mark int  NOT NULL DEFAULT 0,
   CHECK(Mark >= 0 AND Mark <= 100),
   CONSTRAINT Exams_pk PRIMARY KEY  (ExamID)
);
```

```sql
-- Table: Modules
-- Zbiór zajęć na określony temat, nie tożsamy z pojęciem przedmiotu (jeden moduł
może zawierać zajęcia z różnych przedmiotów). Pozwalają na łączenie zajęć różnej
formy kształcenia (stacjonarne, online asynchroniczne, online synchroniczne,
hybrydowe).
-- Dla przykładu:
-- Moduł “Programowanie w matematyce” mógłby obejmować szereg zajęć z przedmiotów
matematycznych, na których problemy rozwiązywane są przy pomocy pisanego kodu
CREATE TABLE Modules (
   ModuleID int  NOT NULL,
   ProgramID int  NOT NULL,
   ModuleName varchar(40)  NOT NULL,
   ModuleDescription varchar(100)  NOT NULL,
   CONSTRAINT Modules_pk PRIMARY KEY  (ModuleID)
);


-- Table: OfflineClasses
-- Podzbiór Classes: pojedyncze zajęcia, prowadzone w trybie offline
(stacjonarnie), zawsze są podporządkowane jednemu modułu zajęć.
CREATE TABLE OfflineClasses (
   OfflineClassID int  NOT NULL,
   ClassID int  NOT NULL,
   RoomNumber int  NOT NULL,
   MaxParticipants int  NOT NULL DEFAULT 0,
   Mark int  NULL,
   CHECK(Mark >= 0 AND Mark <= 100),
   CONSTRAINT OfflineClasses_pk PRIMARY KEY  (OfflineClassID)
);


-- Table: OnlineClasses
-- Podzbiór Classes: pojedyncze zajęcia, prowadzone w trybie online. Obejmują
synchroniczne i asynchroniczne moduły.
CREATE TABLE OnlineClasses (
   OnlineClassID int  NOT NULL,
   ClassID int  NOT NULL,
   Link varchar(255)  NOT NULL,
   Synch bit  NOT NULL DEFAULT 0,
   CONSTRAINT OnlineClasses_pk PRIMARY KEY  (OnlineClassID)
);
```

```sql
-- Table: Orders
-- Lista zamówień przez Studentów. Informacja o zakupionych programach oraz
pojedynczych spotkaniach znajduje się w tabelach RegisteredPrograms i
RegisteredClasses odpowiednio.
CREATE TABLE Orders (
   OrderID int  NOT NULL,
   StudentID int  NOT NULL,
   OrderDate datetime  NOT NULL DEFAULT GETDATE(),
   OrderStatus varchar(40) NOT NULL DEFAULT 'NOT PAID',
   CHECK(OrderStatus IN ('NOT PAID', 'ENTRY PAID', 'FULL PAID'))
   CONSTRAINT Orders_pk PRIMARY KEY  (OrderID)
);


-- Table: Payments
-- Spis płatność dokonanych w celu częściowego lub całkowitego opłacenia
zamówienia z tabeli Orders. Kolumna Status informuje czy płatność została
zakończona sukcesem, natomiast kolumna SystemPaymentID zawiera link do
zewnętrznego systemu płatności.
CREATE TABLE Payments (
   PaymentID int  NOT NULL,
   OrderID int  NOT NULL,
   Amount money  NOT NULL,
   Date date  NOT NULL DEFAULT GETDATE(),
   Status bit  NOT NULL DEFAULT 0,
   CHECK (Amount >= 0),
   CONSTRAINT Payments_pk PRIMARY KEY  (PaymentID)
);


-- Table: Practises
-- Każde studia mogą zawierać wiele praktyk, tabela przetrzymuje opis i
identyfikator danych praktyk. W Tabeli Classes znajduje się pole PracticeID, które
nie jest NULL-em w przypadku gdy dane zajęcia realizują dane praktyki.
CREATE TABLE Practises (
   PractiseID int  NOT NULL,
   StudiesID int  NOT NULL,
   PracticeName varchar(40)  NOT NULL,
   PracticeDescription varchar(255)  NOT NULL,
   CONSTRAINT Practices_pk PRIMARY KEY  (PractiseID)
);
```

```sql
-- Table: RegisteredClasses
-- Lista zakupionych przez studentów pojedynczych classes (zjazdów w ramach
studiów) z numerami zamówienia
CREATE TABLE RegisteredClasses (
   RegisteredClassID int  NOT NULL,
   OrderID int  NOT NULL,
   ClassID int  NOT NULL,
   Access bit NOT NULL DEFAULT 0,
   CONSTRAINT RegisteredClasses_pk PRIMARY KEY  (RegisteredClassID)
);


-- Table: RegisteredPrograms
-- Lista zakupionych przez studentów EducationalProgramów z numerami zamówienia
CREATE TABLE RegisteredPrograms (
   RegisteredProgramID int  NOT NULL,
   OrderID int  NOT NULL,
   ProgramID int  NOT NULL,
   Passed bit  NOT NULL DEFAULT 0,
   CertificateLink varchar(255),
   Access bit NOT NULL DEFAULT 0,
   CONSTRAINT RegisteredPrograms_pk PRIMARY KEY  (RegisteredProgramID)
);


-- Table: Students
CREATE TABLE Students (
   StudentID int  NOT NULL,
   FirstName varchar(20)  NOT NULL,
   LastName varchar(20)  NOT NULL,
   CountryID int  NOT NULL,
   Email varchar(40)  NOT NULL UNIQUE,
   CONSTRAINT Students_pk PRIMARY KEY  (StudentID)
);


-- Table: Studies
CREATE TABLE Studies (
   StudiesID int  NOT NULL,
   Syllabus varchar(255)  NOT NULL,
   Place varchar(100)  NOT NULL,
   MaxParticipants int  NOT NULL,
   EntryFee money  NOT NULL
   CHECK (EntryFee >= 0),
   CONSTRAINT Studies_pk PRIMARY KEY  (StudiesID)
);
```

```sql
-- Table: SubjectCategories
-- Zawiera kategorie różnych prowadzonych przedmiotów z tabeli Subjects
-- np. Matematyka(SubjectCategories) jest kategorią przedmiotu algebra(Subjects)
CREATE TABLE SubjectCategories (
   CategoryID int  NOT NULL,
   CategoryName varchar(40)  NOT NULL UNIQUE,
   Description varchar(255)  NOT NULL,
   CONSTRAINT SubjectCategories_pk PRIMARY KEY  (CategoryID)
);


-- Table: Subjects
CREATE TABLE Subjects (
   SubjectID int  NOT NULL,
   CategoryID int  NOT NULL,
   Description varchar(255)  NOT NULL,
   SubjectName varchar(40)  NOT NULL UNIQUE,
   CONSTRAINT Subjects_pk PRIMARY KEY  (SubjectID)
);


-- Table: Teachers
CREATE TABLE Teachers (
   TeacherID int  NOT NULL,
   FirstName varchar(15)  NOT NULL,
   LastName varchar(15)  NOT NULL,
   CountryID int  NOT NULL,
   CONSTRAINT Teachers_pk PRIMARY KEY  (TeacherID)
);


-- Table: Webinars
CREATE TABLE Webinars (
   WebinarID int  NOT NULL,
   ClassID int  NOT NULL,
   CONSTRAINT Webinars_pk PRIMARY KEY  (WebinarID)
);


-- foreign keys
-- Reference:  Translators_Countries (table:  Translators)
ALTER TABLE  Translators ADD CONSTRAINT  Translators_Countries
   FOREIGN KEY (CountryID)
   REFERENCES Countries (CountryID);


-- Reference: Attendance_Students (table: Attendance)
ALTER TABLE Attendance ADD CONSTRAINT Attendance_Students
   FOREIGN KEY (ParticipantID)
   REFERENCES Students (StudentID);
```

```
-- Reference: Classes_Attendance (table: Attendance)
ALTER TABLE Attendance ADD CONSTRAINT Classes_Attendance
   FOREIGN KEY (ClassID)
   REFERENCES Classes (ClassID);


-- Reference: Classes_Modules (table: Classes)
ALTER TABLE Classes ADD CONSTRAINT Classes_Modules
   FOREIGN KEY (ModuleID)
   REFERENCES Modules (ModuleID);


-- Reference: Classes_Practises (table: Classes)
ALTER TABLE Classes ADD CONSTRAINT Classes_Practises
   FOREIGN KEY (PractiseID)
   REFERENCES Practises (PractiseID);


-- Reference: Classes_Subjects (table: Classes)
ALTER TABLE Classes ADD CONSTRAINT Classes_Subjects
   FOREIGN KEY (SubjectID)
   REFERENCES Subjects (SubjectID);


-- Reference: Classes_Teachers (table: Classes)
ALTER TABLE Classes ADD CONSTRAINT Classes_Teachers
   FOREIGN KEY (TeacherID)
   REFERENCES Teachers (TeacherID);


-- Reference: EducationalPrograms_ Translators (table: EducationalPrograms)
ALTER TABLE EducationalPrograms ADD CONSTRAINT EducationalPrograms_Translators
   FOREIGN KEY (TranslatorID)
   REFERENCES  Translators (TranslatorID);


-- Reference: EducationalPrograms_Courses (table: EducationalPrograms)
ALTER TABLE EducationalPrograms ADD CONSTRAINT EducationalPrograms_Courses
   FOREIGN KEY (CourseID)
   REFERENCES Courses (CourseID);


-- Reference: EducationalPrograms_Studies (table: EducationalPrograms)
ALTER TABLE EducationalPrograms ADD CONSTRAINT EducationalPrograms_Studies
   FOREIGN KEY (StudiesID)
   REFERENCES Studies (StudiesID);


-- Reference: EducationalPrograms_Teachers (table: EducationalPrograms)
ALTER TABLE EducationalPrograms ADD CONSTRAINT EducationalPrograms_Teachers
   FOREIGN KEY (LecturerID)
   REFERENCES Teachers (TeacherID);
```

```sql
-- Reference: EducationalPrograms_Webinars (table: EducationalPrograms)
ALTER TABLE EducationalPrograms ADD CONSTRAINT EducationalPrograms_Webinars
   FOREIGN KEY (WebinarID)
   REFERENCES Webinars (WebinarID);


-- Reference: Exams_Students (table: Exams)
ALTER TABLE Exams ADD CONSTRAINT Exams_Students
   FOREIGN KEY (StudentID)
   REFERENCES Students (StudentID);


-- Reference: Exams_Studies (table: Exams)
ALTER TABLE Exams ADD CONSTRAINT Exams_Studies
   FOREIGN KEY (StudiesID)
   REFERENCES Studies (StudiesID);


-- Reference: Modules_EducationalPrograms (table: Modules)
ALTER TABLE Modules ADD CONSTRAINT Modules_EducationalPrograms
   FOREIGN KEY (ProgramID)
   REFERENCES EducationalPrograms (ProgramID);


-- Reference: OfflineClasses_Classes (table: OfflineClasses)
ALTER TABLE OfflineClasses ADD CONSTRAINT OfflineClasses_Classes
   FOREIGN KEY (ClassID)
   REFERENCES Classes (ClassID);


-- Reference: OnlineClasses_Classes (table: OnlineClasses)
ALTER TABLE OnlineClasses ADD CONSTRAINT OnlineClasses_Classes
   FOREIGN KEY (ClassID)
   REFERENCES Classes (ClassID);


-- Reference: OrderClasses_Classes (table: RegisteredClasses)
ALTER TABLE RegisteredClasses ADD CONSTRAINT OrderClasses_Classes
   FOREIGN KEY (ClassID)
   REFERENCES Classes (ClassID);


-- Reference: OrderClasses_Orders (table: RegisteredClasses)
ALTER TABLE RegisteredClasses ADD CONSTRAINT OrderClasses_Orders
   FOREIGN KEY (OrderID)
   REFERENCES Orders (OrderID);

-- Reference: OrderDetails_EducationalPrograms (table: RegisteredPrograms)
ALTER TABLE RegisteredPrograms ADD CONSTRAINT OrderDetails_EducationalPrograms
   FOREIGN KEY (ProgramID)
   REFERENCES EducationalPrograms (ProgramID);


-- Reference: OrderDetails_Orders (table: RegisteredPrograms)
```

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


-- Reference: Orders_Students (table: Orders)
ALTER TABLE Orders ADD CONSTRAINT Orders_Students
   FOREIGN KEY (StudentID)
   REFERENCES Students (StudentID);


-- Reference: Payments_Orders (table: Payments)
ALTER TABLE Payments ADD CONSTRAINT Payments_Orders
   FOREIGN KEY (OrderID)
   REFERENCES Orders (OrderID);


-- Reference: Practises_Studies (table: Practises)
ALTER TABLE Practises ADD CONSTRAINT Practises_Studies
   FOREIGN KEY (StudiesID)
   REFERENCES Studies (StudiesID);


-- Reference: Students_Countries (table: Students)
ALTER TABLE Students ADD CONSTRAINT Students_Countries
   FOREIGN KEY (CountryID)
   REFERENCES Countries (CountryID);


-- Reference: Subjects_SubjectCategories (table: Subjects)
ALTER TABLE Subjects ADD CONSTRAINT Subjects_SubjectCategories
   FOREIGN KEY (CategoryID)
   REFERENCES SubjectCategories (CategoryID);


-- Reference: Teachers_Countries (table: Teachers)
ALTER TABLE Teachers ADD CONSTRAINT Teachers_Countries
   FOREIGN KEY (CountryID)
   REFERENCES Countries (CountryID);


-- Reference: Webinars_Classes (table: Webinars)
ALTER TABLE Webinars ADD CONSTRAINT Webinars_Classes
   FOREIGN KEY (ClassID)
   REFERENCES Classes (ClassID);
```

3. **Widoki**

1. Raporty finansowe – zestawienie przychodów dla każdego webinaru/kursu/studium

```sql
-- Webinars
CREATE VIEW WebinarsRevenue
AS(
SELECT Webinars.WebinarID, EducationalPrograms.ProgramName, SUM(Payments.Amount)
AS Revenue
FROM Webinars
JOIN EducationalPrograms ON Webinars.WebinarID = EducationalPrograms.WebinarID
JOIN RegisteredPrograms ON RegisteredPrograms.ProgramID =
EducationalPrograms.ProgramID
JOIN Orders ON Orders.OrderID = RegisteredPrograms.OrderID
JOIN Payments ON Orders.OrderID = Payments.OrderID
GROUP BY Webinars.WebinarID, EducationalPrograms.ProgramName;
)

--Courses
CREATE VIEW CoursesRevenue
AS(
SELECT Courses.CourseID, EducationalPrograms.ProgramName, SUM(Payments.Amount) AS
Revenue
FROM Courses
INNER JOIN EducationalPrograms ON Courses.CourseID = EducationalPrograms.CourseID
JOIN RegisteredPrograms ON RegisteredPrograms.ProgramID =
EducationalPrograms.ProgramID
JOIN Orders ON Orders.OrderID = RegisteredPrograms.OrderID
JOIN Payments ON Orders.OrderID = Payments.OrderID
GROUP BY Courses.CourseID, EducationalPrograms.ProgramName;
)

-- Studies
CREATE VIEW StudentRevenue
AS
(
SELECT Studies.StudiesID, EducationalPrograms.ProgramName, SUM(Payments.Amount) AS
Revenue
FROM Studies
JOIN EducationalPrograms ON Studies.StudiesID = EducationalPrograms.StudiesID
JOIN RegisteredPrograms ON RegisteredPrograms.ProgramID =
EducationalPrograms.ProgramID
JOIN Modules ON Modules.ProgramID = EducationalPrograms.ProgramID
JOIN Practises ON Practises.StudiesID = Studies.StudiesID
JOIN Classes ON Classes.ModuleID = Modules.ModuleID OR Classes.PractiseID =
Practises.PractiseID
JOIN RegisteredClasses ON RegisteredClasses.ClassID = Classes.ClassID
JOIN Orders ON Orders.OrderID = RegisteredClasses.OrderID OR Orders.OrderID =
RegisteredPrograms.OrderID
JOIN Payments ON Payments.OrderID = Orders.OrderID
GROUP BY Studies.StudiesID, EducationalPrograms.ProgramName;
)
```

2. Lista „dłużników” – osoby, które skorzystały z usług, ale nie uiściły opłat.

```
CREATE VIEW Debtors
AS(
SELECT S.*, ISNULL(OPS.ProgramsCost, 0) + ISNULL(OCS.ClassesCost, 0) -
ISNULL(OP.Paid, 0) As Debt
FROM Students S
LEFT JOIN (
    SELECT O.StudentID, SUM(EP.ProgramPrice) AS ProgramsCost
    FROM Orders O
    LEFT JOIN RegisteredPrograms RP ON RP.OrderID = O.OrderID
    LEFT JOIN EducationalPrograms EP ON EP.ProgramID = RP.ProgramID
    GROUP BY O.StudentID
) OPS ON OPS.StudentID = S.StudentID
LEFT JOIN (
    SELECT O.StudentID, SUM(C.ClassPrice) AS ClassesCost
    FROM Orders O
    LEFT JOIN RegisteredClasses RC ON RC.OrderID = O.OrderID
    LEFT JOIN Classes C ON C.ClassID = RC.ClassID
    GROUP BY O.StudentID
) OCS ON OCS.StudentID = S.StudentID
LEFT JOIN (
    SELECT O.StudentID, SUM(P.Amount) AS Paid
    FROM Orders O
    LEFT JOIN Payments P ON P.OrderID = O.OrderID
    GROUP BY O.StudentID
) OP ON OP.StudentID = S.StudentID
WHERE ISNULL(OPS.ProgramsCost, 0) + ISNULL(OCS.ClassesCost, 0) > ISNULL(OP.Paid,
0)
)
```

4. Ogólny raport dotyczący frekwencji na zakończonych już wydarzeniach.

```
-- a) Lista osób
CREATE VIEW  ParticipantsList
AS(
select a.ClassID, s.StudentID, s.FirstName + ' ' + s.LastName as Student,
c.TeacherID, c.SubjectID, c.StartTime, c.EndTime
    from Attendance as a
        inner join Students as s
            on a.ParticipantID = s.StudentID
        inner join Classes as c
            on a.ClassID = c.ClassID
    where c.EndTime < getdate()
)
```

```
-- b) Liczba osób dla każdego wydarzenia
CREATE VIEW NumberOfParticipants AS
(
select a.ClassID, c.TeacherID, c.SubjectID, c.StartTime, c.EndTime,
count(s.StudentID) as StudentsAmount
    from Classes as c
        left join Attendance as a
            on c.ClassID = a.ClassID
        inner join Students as s
            on a.ParticipantID = s.StudentID
    where c.EndTime < getdate()
group by a.ClassID, c.TeacherID, c.SubjectID, c.StartTime, c.EndTime
)
```

5. Lista obecności dla każdego szkolenia z datą, imieniem, nazwiskiem i informacją czy uczestnik był obecny, czy nie.

```
CREATE VIEW AttendanceAllClasses
AS(
SELECT C.ClassID, CONCAT(year(C.StartTime), '-', month(C.StartTime), '-',
day(C.StartTime)) as Date, Students.FirstName + ' ' + Students.LastName as
Student, A.Present
FROM Classes C
LEFT OUTER JOIN Attendance A on C.ClassID = A.ClassID
LEFT OUTER JOIN Students on Students.StudentID = A.ParticipantID
)
```

6. Raport bilokacji: lista osób, które są zapisane na co najmniej dwa przyszłe szkolenia, które ze sobą kolidują czasowo.

```
CREATE VIEW BilocationsList
AS(
select distinct s.StudentID, s.FirstName + ' ' + s.LastName as Student, a.ClassID
as a_ClassID, a.StartTime as a_StartTime, a.EndTime as a_EndTime, b.ClassID as
b_ClassID, b.StartTime as b_StartTime, b.EndTime as b_EndTime from Students as s
        inner join Orders as o
            on s.StudentID = o.StudentID
        inner join RegisteredPrograms as rp
            on o.OrderID = rp.OrderID
        inner join Modules as m
            on rp.ProgramID = m.ProgramID
        inner join Classes as a
            on m.ModuleID = a.ModuleID
        cross join Classes as b
where a.ClassID < b.ClassID and ((a.StartTime BETWEEN b.StartTime and b.EndTime)
or (b.StartTime BETWEEN a.StartTime and a.EndTime) or (a.EndTime BETWEEN
b.StartTime and b.EndTime) or (b.EndTime BETWEEN a.StartTime and a.EndTime))
)
```

4. **Procedury**

1. Dodanie nowego Studenta

```
CREATE PROCEDURE AddStudent(
   @firstName VARCHAR(20),
   @lastName VARCHAR(20),
   @countryID INT,
   @email VARCHAR(40)
)
AS
BEGIN
   BEGIN TRY
       IF EXISTS(SELECT * FROM Students WHERE Email = @email)
            THROW 52034, N'Email already in use', 1;
       ELSE
            INSERT INTO Students (FirstName, LastName, CountryID, Email)
            VALUES (@firstName, @lastName, @countryID, @email);
   END TRY
   BEGIN CATCH
       DECLARE @Message NVARCHAR(1000) = N'error: ' + ERROR_MESSAGE();
       THROW 52034, @Message, 1;
   END CATCH
END
```

2. Dodanie nowego kursu

```
CREATE PROCEDURE AddCourse
   @Place varchar(40),
   @Advance money,
   @ProgramName varchar(100),
   @Language varchar(20),
   @ProgramStart date,
   @ProgramEnd date,
   @ProgramPrice money,
   @LecturerID int,
   @TranslatorID int
AS
BEGIN
BEGIN TRY
   SET NOCOUNT ON;

   IF NOT EXISTS (SELECT 1 FROM Teachers WHERE TeacherID = @LecturerID)
   BEGIN
       THROW 50000, 'TeacherID does not exist in the Teachers table.', 1;
   END;


   IF NOT EXISTS (SELECT 1 FROM Translators WHERE TranslatorID = @TranslatorID)
   BEGIN
       THROW 50000, 'TranslatorID does not exist in the Translators table.', 1;
   END;

   INSERT INTO Courses (Place, Advance)
   VALUES (@Place, @Advance);


   INSERT INTO EducationalPrograms (ProgramName, CourseID, Language, ProgramStart,
ProgramEnd, ProgramPrice, LecturerID, TranslatorID)
   VALUES (@NewProgramID, @ProgramName, @NewCourseID, @Language, @ProgramStart,
@ProgramEnd, @ProgramPrice, @LecturerID, @TranslatorID);

END TRY
BEGIN CATCH
    DECLARE @Message NVARCHAR(1000) = N'error: ' + ERROR_MESSAGE();
    THROW 52011, @Message, 1;
END CATCH
END;
```

3. Usuwanie studenta

```sql
ALTER PROCEDURE DeleteStudent(@studentID INT)
AS
BEGIN
   BEGIN TRY
       IF NOT EXISTS(
               SELECT *
               FROM Students
               WHERE StudentID = @studentID
           )
           BEGIN
               THROW 52000, N'There is no student with given ID', 1;
           END
           DECLARE @an NVARCHAR(10) = 'xxxxxxxx'
           UPDATE Students
               SET FirstName    = @an,
               LastName     = @an,
               Email      = @an
           WHERE StudentID = @studentID
   END TRY
   BEGIN CATCH
       DECLARE @msg NVARCHAR(2048) = N'ERROR: ' + ERROR_MESSAGE();
       THROW 52000, @msg, 1;
   END CATCH
END
```

4. Aktualizacja danych studenta (tylko email albo country)

```sql
CREATE PROCEDURE ChangeStudentData(
  @studentID int,
  @countryID int = NULL,
  @email varchar(40) = NULL)
AS
BEGIN
   SET NOCOUNT ON;
   IF @countryID IS NOT NULL
   BEGIN
       IF @countryID in (select CountryID from Countries)
           UPDATE Students SET CountryID = @countryID WHERE StudentID = @studentID
   END
   IF @email IS NOT NULL
   BEGIN
       UPDATE Students SET Email = @Email WHERE StudentID = @studentID
   END
END
```

5. Dodanie nowego nauczyciela

```
CREATE PROCEDURE AddTeacher(
  @firstName VARCHAR(20),
  @lastName VARCHAR(20),
  @countryID INT
)
AS
BEGIN

  INSERT INTO Teachers (FirstName, LastName, CountryID)
  VALUES (@firstName, @lastName, @countryID)
END
```

**6. Oznaczenie odrobienia nieobecności na zajęciach przez studenta **

```


create procedure redoAttendance @classID int, @studentID int
as
begin
   set nocount on
   begin try
       if not exists
           (
           select ClassID, ParticipantID
               from Attendance
           WHERE ClassID = @classID and ParticipantID = @studentID
           )
       begin;
           throw 52000, N'The student was not registered for the class with the
given ID', 1
       end
       if exists
           (
           select ClassID, ParticipantID
               from Attendance
           WHERE ClassID = @classID and ParticipantID = @studentID and Redone = 1
           )
       begin;
           throw 52000, N'Attendance had already been made up earlier', 1
       end
       update Attendance
       set Redone = 1
       where ClassID = @classID and ParticipantID = @studentID
       print 'Attendance was successfully set as redone!'
   end try
   begin catch
       declare @error varchar(1000)= 'Error when setting attendance as made up: '
+ ERROR_MESSAGE();
       throw 77777, @error, 1
```

```
    end catch
end
```

8. Dodanie pojedynczych zajęć do zamówienia

```sql
CREATE PROCEDURE RegisterClass(
  @OrderID INT,
  @ClassID INT
)
AS
BEGIN
 BEGIN TRY
      IF NOT EXISTS(SELECT * FROM Orders WHERE OrderID = @OrderID)
       THROW 52313, N'There is no Order with such id', 1;

      IF NOT EXISTS(SELECT * FROM Classes WHERE ClassID = @ClassID)
       THROW 52313, N'There is no Class with such id', 1;


      INSERT INTO RegisteredClasses (OrderID, ClassID)
      VALUES (@OrderID, @ClassID);
      SELECT 'Class added successfully.' AS Message;
 END TRY
 BEGIN CATCH
     DECLARE @Message NVARCHAR(1000) = N'error: ' + ERROR_MESSAGE();
     THROW 52011, @Message, 1;
 END CATCH
END
```

9. Dodawanie programu edukacyjnego do zamówienia

```
CREATE PROCEDURE RegisterProgram(
  @OrderID INT,
  @ProgramID INT,
  @Passed AS bit = FALSE,
  @CertificateLink AS varchar(255) = NULL
)
AS
BEGIN
 BEGIN TRY
      IF NOT EXISTS(SELECT * FROM Orders WHERE OrderID = @OrderID)
       THROW 52313, N'There is no Order with such id', 1;


      IF NOT EXISTS(SELECT * FROM EducationalPrograms WHERE ProgramID =
@ProgramID)
       THROW 52313, N'There is no EducationlProram with such id', 1;


      INSERT INTO RegisteredPrograms (OrderID, ProgramID, Passed, CertificateLink)
      VALUES (@OrderID, @ProgramID, @Passed, @CertificateLink);
      SELECT 'Program added successfully.' AS Message;
 END TRY
 BEGIN CATCH
     DECLARE @Message NVARCHAR(1000) = N'error: ' + ERROR_MESSAGE();
     THROW 52011, @Message, 1;
 END CATCH
END;
```

10. Dodanie nowego pojedynczego niestacjonarnego zajęcia

```
CREATE PROCEDURE AddOnlineClass
 @Link varchar(255),
 @Synch bit,
 @TeacherID int,
 @SubjectID int,
 @StartTime datetime,
 @EndTime datetime,
 @ClassPrice money = NULL,
 @ModuleID int = NULL,
 @PractiseID int = NULL,
 @NewClassID int OUTPUT

AS
BEGIN
 SET NOCOUNT ON;
 DECLARE @NewOnlineClassID int;
 BEGIN TRY
     IF NOT EXISTS (SELECT 1 FROM Teachers WHERE TeacherID = @TeacherID)
     BEGIN
         THROW 50000, 'TeacherID does not exist in the Teachers table.', 1;
     END;
     IF NOT EXISTS (SELECT 1 FROM Subjects WHERE SubjectID = @SubjectID)
     BEGIN
         THROW 50000, 'SubjectID does not exist in the Subjects table.', 1;
     END;
     IF (@ModuleID IS NOT NULL AND @PractiseID IS NOT NULL)
     BEGIN
         THROW 50000, 'Can`t define both ModuleID and PractiseID', 1;
     END;
     IF @ModuleID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Modules WHERE ModuleID
= @ModuleID)
     BEGIN
         THROW 50000, 'ModuleID does not exist in the Modules table.', 1;
     END;
     IF @PractiseID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Practises WHERE
PractiseID = @PractiseID)
     BEGIN
         THROW 50000, 'PractiseID does not exist in the Practises table.', 1;
     END;

     SELECT @NewClassID = ISNULL(MAX(ClassID), 0) + 1
     FROM Classes;

     INSERT INTO Classes (ClassID, TeacherID, SubjectID, StartTime, EndTime,
ClassPrice)
     VALUES (@NewClassID, @TeacherID, @SubjectID, @StartTime, @EndTime,
@ClassPrice);
```

```
        SELECT @NewOnlineClassID = ISNULL(MAX(OnlineClassID), 0) + 1
        FROM OnlineClasses;

        INSERT INTO OnlineClasses (OnlineClassID, ClassID, Link, Synch)
        VALUES (@NewOnlineClassID, @NewClassID, @Link, @Synch)
        IF @ModuleID IS NOT NULL

        BEGIN
            UPDATE Classes SET ModuleID = @ModuleID WHERE ClassID = @NewClassID
        END;
        IF @PractiseID IS NOT NULL
        BEGIN
            UPDATE Classes SET PractiseID = @PractiseID WHERE ClassID = @NewClassID
        END;
        PRINT 'OnlineClass added successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @msg NVARCHAR(2048) = N'ERROR: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END;
```

11. Dodanie nowego pojedynczego stacjonarnego zajęcia

```


CREATE PROCEDURE AddOfflineClass
  @RoomNumber int,
  @MaxParticipants int,
  @TeacherID int,
  @SubjectID int,
  @StartTime datetime,
  @EndTime datetime,
  @ClassPrice money = NULL,
  @ModuleID int,
  @PractiseID int = NULL,
  @NewClassID int OUTPUT


AS
BEGIN
  SET NOCOUNT ON;

  BEGIN TRY
      IF NOT EXISTS (SELECT 1 FROM Teachers WHERE TeacherID = @TeacherID)
      BEGIN
          THROW 50000, 'TeacherID does not exist in the Teachers table.', 1;
      END;


      IF NOT EXISTS (SELECT 1 FROM Subjects WHERE SubjectID = @SubjectID)
```

```
        BEGIN
            THROW 50000, 'SubjectID does not exist in the Subjects table.', 1;
        END;


        IF @ModuleID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Modules WHERE
ModuleID = @ModuleID)
        BEGIN
            THROW 50000, 'ModuleID does not exist in the Modules table.', 1;
        END;


        IF @PractiseID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Practises WHERE
PractiseID = @PractiseID)
        BEGIN
            THROW 50000, 'PractiseID does not exist in the Practises table.', 1;
        END;


        INSERT INTO Classes (TeacherID, SubjectID, StartTime, EndTime, ClassPrice,
ModuleID)
        VALUES (@TeacherID, @SubjectID, @StartTime, @EndTime, @ClassPrice,
@ModuleID);


        INSERT INTO OfflineClasses (ClassID, RoomNumber, MaxParticipants)
        VALUES (@NewClassID, @RoomNumber, @MaxParticipants)


        IF @PractiseID IS NOT NULL
        BEGIN
            UPDATE Classes SET PractiseID = @PractiseID WHERE ClassID = @NewClassID
        END;


        PRINT 'OfflineClass added successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @msg NVARCHAR(2048) = N'ERROR: ' + ERROR_MESSAGE();
        THROW 52000, @msg, 1;
    END CATCH
END;
```

12. Dodanie nowego webinaru

```
CREATE PROCEDURE AddWebinar
  @ProgramName varchar(100),
  @Language varchar(20),
  @ProgramStart date,
  @ProgramEnd date,
  @ProgramPrice money,
  @LecturerID int,
  @TranslatorID int = NULL,


  @Link varchar(255),
  @Synch bit,
  @SubjectID int,
  @StartTime datetime,
  @EndTime datetime,
  @ClassPrice money = NULL,
  @ModuleID int = NULL,
  @PractiseID int = NULL


AS
BEGIN
  SET NOCOUNT ON;


  DECLARE @NewClassID int;
  DECLARE @NewWebinarID int;
  DECLARE @NewProgramID int;


  IF NOT EXISTS (SELECT 1 FROM Teachers WHERE TeacherID = @LecturerID)
  BEGIN
      THROW 50000, 'LecturerID does not exist in the Teachers table.', 1;
  END;


  IF @TranslatorID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Translators WHERE
TranslatorID = @TranslatorID)
  BEGIN
      THROW 50000, 'TranslatorID does not exist in the Translators table.', 1;
  END;
  EXEC AddOnlineClass @Link, @Synch, @LecturerID, @SubjectID, @StartTime,
@EndTime, @ClassPrice, @ModuleID, @PractiseID, @NewClassID OUTPUT


  SELECT @NewWebinarID = ISNULL(MAX(WebinarID), 0) + 1
  FROM Webinars;


  INSERT INTO Webinars (WebinarID, ClassID)
  VALUES (@NewWebinarID, @NewClassID);


  SELECT @NewProgramID = ISNULL(MAX(ProgramID), 0) + 1
  FROM EducationalPrograms;


  INSERT INTO EducationalPrograms (ProgramID, ProgramName, WebinarID, Language,
ProgramStart, ProgramEnd, ProgramPrice, LecturerID, TranslatorID)
```

```
  VALUES (@NewProgramID, @ProgramName, @NewWebinarID, @Language, @ProgramStart,
@ProgramEnd, @ProgramPrice, @LecturerID, @TranslatorID);



  PRINT 'Webinar added successfully.';
END;
```

13. Dodanie nowego kursu

```
CREATE PROCEDURE AddCourse
  @ProgramName varchar(100),
  @Language varchar(20),
  @ProgramStart date,
  @ProgramEnd date,
  @ProgramPrice money,
  @LecturerID int,
  @TranslatorID int = NULL,
  @Place varchar(20),
  @Advance money


AS
BEGIN
  SET NOCOUNT ON;


  DECLARE @NewCourseID int;
  DECLARE @NewProgramID int;


  IF NOT EXISTS (SELECT 1 FROM Teachers WHERE TeacherID = @LecturerID)
  BEGIN
      THROW 50000, 'LecturerID does not exist in the Teachers table.', 1;
  END;


  IF @TranslatorID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Translators WHERE
TranslatorID = @TranslatorID)
  BEGIN
      THROW 50000, 'TranslatorID does not exist in the Translators table.', 1;
  END;


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


  INSERT INTO Courses (CourseID, Place, Advance)
  VALUES (@NewCourseID, @Place, @Advance);


  SELECT @NewProgramID = ISNULL(MAX(ProgramID), 0) + 1
  FROM EducationalPrograms;


  INSERT INTO EducationalPrograms (ProgramID, ProgramName, CourseID, Language,
ProgramStart, ProgramEnd, ProgramPrice, LecturerID, TranslatorID)
  VALUES (@NewProgramID, @ProgramName, @NewCourseID, @Language, @ProgramStart,
@ProgramEnd, @ProgramPrice, @LecturerID, @TranslatorID);




  PRINT 'Course added successfully.';
END;
```

14. Dodanie nowych studiów

```


CREATE PROCEDURE AddStudies
  @ProgramName varchar(100),
  @Language varchar(20),
  @ProgramStart date,
  @ProgramEnd date,
  @ProgramPrice money,
  @LecturerID int,
  @TranslatorID int = NULL,
  @Syllabus varchar(255),
  @Place varchar(20),
  @MaxParticipants int,
  @EntryFee money


AS
BEGIN
  SET NOCOUNT ON;


  DECLARE @NewStudiesID int;
  DECLARE @NewProgramID int;


  IF NOT EXISTS (SELECT 1 FROM Teachers WHERE TeacherID = @LecturerID)
  BEGIN
      THROW 50000, 'LecturerID does not exist in the Teachers table.', 1;
  END;


  IF @TranslatorID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Translators WHERE
TranslatorID = @TranslatorID)
  BEGIN
      THROW 50000, 'TranslatorID does not exist in the Translators table.', 1;
  END;


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


  INSERT INTO Studies (StudiesID, Syllabus, Place, MaxParticipants, EntryFee)
  VALUES (@NewStudiesID, @Syllabus, @Place, @MaxParticipants, @EntryFee);


  SELECT @NewProgramID = ISNULL(MAX(ProgramID), 0) + 1
  FROM EducationalPrograms;


  INSERT INTO EducationalPrograms (ProgramID, ProgramName, StudiesID, Language,
ProgramStart, ProgramEnd, ProgramPrice, LecturerID, TranslatorID)
  VALUES (@NewProgramID, @ProgramName, @NewStudiesID, @Language, @ProgramStart,
@ProgramEnd, @ProgramPrice, @LecturerID, @TranslatorID);




  PRINT 'Studies added successfully.';
END;
```

15. Zmiana szczegółów programu edukacyjnego

```
CREATE PROCEDURE UpdateEducationalProgram
   @ProgramID INT,
   @NewProgramName VARCHAR(100) = NULL,
   @NewLanguage VARCHAR(20) = NULL,
   @NewProgramStart DATE = NULL,
   @NewProgramEnd DATE = NULL,
   @NewProgramPrice MONEY = NULL,
   @NewLecturerID INT = NULL,
   @NewTranslatorID INT = NULL,
   @NewSyllabus VARCHAR(255) = NULL,
   @NewStudiesPlace VARCHAR(100) = NULL,
   @NewMinParticipants INT = NULL,
   @NewEntryFee MONEY = NULL,
   @NewCoursesPlace VARCHAR(40) = NULL,
   @NewAdvance MONEY = NULL,
   @NewClassID INT = NULL
AS
BEGIN
 BEGIN TRY
   SET NOCOUNT ON;
   DECLARE @IsCourse BIT, @IsWebinar BIT, @IsStudies BIT


   -- Sprawdzenie typu programu na podstawie ProgramID
   SELECT @IsCourse = IIF(EXISTS (SELECT 1 FROM EducationalPrograms WHERE
ProgramID = @ProgramID and CourseID IS NOT NULL), 1, 0),
          @IsWebinar = IIF(EXISTS (SELECT 1 FROM EducationalPrograms WHERE
ProgramID = @ProgramID and WebinarID IS NOT NULL), 1, 0),
          @IsStudies = IIF(EXISTS (SELECT 1 FROM EducationalPrograms WHERE
ProgramID = @ProgramID and StudiesID IS NOT NULL), 1, 0)


   IF @IsStudies = 1 AND (@NewSyllabus IS NOT NULL OR @NewStudiesPlace IS NOT NULL
OR @NewMinParticipants IS NOT NULL OR @NewEntryFee IS NOT NULL)
   BEGIN
       UPDATE Studies
       SET Syllabus = ISNULL(@NewSyllabus, Syllabus),
           Place = ISNULL(@NewStudiesPlace, Place),
           MaxParticipants = ISNULL(@NewMinParticipants, MaxParticipants),
           EntryFee = ISNULL(@NewEntryFee, EntryFee)
       FROM Studies
       JOIN EducationalPrograms ON Studies.StudiesID =
EducationalPrograms.StudiesID
       WHERE EducationalPrograms.ProgramID = @ProgramID;
   END
   ELSE IF @IsStudies = 0 AND (@NewSyllabus IS NOT NULL OR @NewStudiesPlace IS NOT
NULL OR @NewMinParticipants IS NOT NULL OR @NewEntryFee IS NOT NULL)
       THROW 52313, N'EducationalProgram is not ranked in Studies', 10;
   ELSE IF @IsCourse = 1 AND (@NewCoursesPlace IS NOT NULL OR @NewAdvance IS NOT
NULL)
   BEGIN
       UPDATE Courses
       SET Place = ISNULL(@NewCoursesPlace, Place),
           Advance = ISNULL(@NewAdvance, Advance)
       FROM Courses
       JOIN EducationalPrograms ON Courses.CourseID = EducationalPrograms.CourseID
       WHERE EducationalPrograms.ProgramID = @ProgramID;
   END
   ELSE IF @IsCourse = 0 AND (@NewCoursesPlace IS NOT NULL OR @NewAdvance IS NOT
NULL)
       THROW 52313, N'EducationalProgram is not ranked in Courses', 10;


   ELSE IF @IsWebinar = 1 AND @NewClassID IS NOT NULL
   BEGIN
       IF NOT EXISTS (SELECT 1 FROM Classes WHERE ClassID = @NewClassID)
           THROW 52314, N'NewClassID does not exist in Classes', 10;
       UPDATE Webinars
       SET ClassID = @NewClassID
       FROM Webinars
       JOIN EducationalPrograms ON Webinars.WebinarID =
EducationalPrograms.WebinarID
       WHERE EducationalPrograms.ProgramID = @ProgramID;
   END
   ELSE IF @IsWebinar = 0 AND @NewClassID IS NOT NULL
       THROW 52313, N'EducationalProgram is not ranked in Webinars', 10;


   IF (@NewProgramName IS NOT NULL OR @NewLanguage IS NOT NULL OR @NewProgramStart
IS NOT NULL OR
       @NewProgramEnd IS NOT NULL OR @NewProgramPrice IS NOT NULL OR
@NewLecturerID IS NOT NULL OR
       @NewTranslatorID IS NOT NULL)
   BEGIN
       IF @NewLecturerID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Teachers WHERE
TeacherID = @NewLecturerID)
           THROW 52314, N'NewLecturerID does not exist in Teachers', 10;
       IF @NewTranslatorID IS NOT NULL AND NOT EXISTS (SELECT 1 FROM Translators
WHERE TranslatorID = @NewTranslatorID)
           THROW 52314, N'NewTranslatorID does not exist in Translators', 10;
       UPDATE EducationalPrograms
       SET ProgramName = ISNULL(@NewProgramName, ProgramName),
           Language = ISNULL(@NewLanguage, Language),
           ProgramStart = ISNULL(@NewProgramStart, ProgramStart),
           ProgramEnd = ISNULL(@NewProgramEnd, ProgramEnd),
           ProgramPrice = ISNULL(@NewProgramPrice, ProgramPrice),
           LecturerID = ISNULL(@NewLecturerID, LecturerID),
           TranslatorID = ISNULL(@NewTranslatorID, TranslatorID)
       WHERE ProgramID = @ProgramID
   END
 END TRY
```

```
  BEGIN CATCH
    DECLARE @Message NVARCHAR(1000) = N'error: ' + ERROR_MESSAGE();
    THROW 123456, @Message, 10;
  END CATCH
END;
```

16. Dodanie Płatności do złożonego zamówienia

```


CREATE PROCEDURE AddPayment
   @OrderID INT,
   @SystemPaymentID VARCHAR(255),
   @PayFull Bit
AS
BEGIN
   BEGIN TRY
       -- Check if the OrderID exists in the Orders table
       IF EXISTS (SELECT * FROM Orders WHERE OrderID = @OrderID)
       BEGIN
           DECLARE @price INT;
           IF @PayFull = 1
           BEGIN
           SELECT @price = dbo.CalculateFullPriceForOrder(@OrderID)
           END

           ELSE
           BEGIN
           SELECT @price = dbo.CalculateEntryPriceForOrder(@OrderID)
           END

           FROM Payments
           INSERT INTO Payments (OrderId, Amount, Date, Status, SystemPaymentID )
           VALUES (@OrderID, @price, GETDATE(), 0, @SystemPaymentID)
       END
       ELSE
       BEGIN
           THROW 51234, N'There is no order with such ID', 1;
       END
   END TRY


   BEGIN CATCH
       DECLARE @ErrorMessage NVARCHAR(1000) = N'Error: ' + ERROR_MESSAGE();
       THROW 52011, @ErrorMessage, 1;
   END CATCH
END;
```

18. Wyswietl listę obecności dla wydarzenia

```
CREATE PROCEDURE GetAttendanceReport
   @ClassID int
AS
BEGIN
   IF EXISTS ( SELECT * FROM Classes WHERE ClassID = @ClassID)
   BEGIN
       SELECT
           s.FirstName + ' ' + s.LastName AS StudentName,
           a.Present,
           a.Redone
       FROM Attendance a
       JOIN Students s ON a.ParticipantID = s.StudentID
       WHERE a.ClassID = @ClassID;
   END
   ELSE
   BEGIN
       THROW 53523, N'There is no such class', 1;
   END
END;
```

19. Dodanie nowych offline zajęć w ramach studiów

```
CREATE PROCEDURE AddStudiesOfflineClasses
  @StudiesID int,
  @RoomNumber int,
  @MaxParticipants int,
  @TeacherID int,
  @SubjectID int,
  @StartTime datetime,
  @EndTime datetime,
  @ClassPrice money = NULL,
  @ModuleID int,
  @PractiseID int = NULL


AS
BEGIN
  SET NOCOUNT ON;


  DECLARE @StudiesMaxParticipants int;
  DECLARE @NewClassID int;


  SELECT @StudiesMaxParticipants = (
          SELECT MaxParticipants from Studies
          where StudiesID = @StudiesID)
  IF (@StudiesMaxParticipants > @MaxParticipants)
  BEGIN
      THROW 50000, 'Amount of each ClassesMaxParticipants should be equal or
greater than StudiesMaxParticipants.', 1;
  END;


  EXEC AddOfflineClass @RoomNumber, @MaxParticipants, @TeacherID, @SubjectID,
@StartTime, @EndTime, @ClassPrice, @ModuleID, @PractiseID, @NewClassID OUTPUT
END;
```

4. **Funkcje**

1.Obliczanie średniej ocen dla studenta

```
CREATE FUNCTION CalculateAverageGradeForStudent
(
   @StudentID int
)
RETURNS DECIMAL(5, 2)
AS
BEGIN
   DECLARE @AverageGrade DECIMAL(5, 2);


   SELECT @AverageGrade = AVG(Mark)
   FROM Exams
   WHERE StudentID = @StudentID;


   RETURN ISNULL(@AverageGrade, 0); -- Jeśli nie ma ocen, zwraca 0
END;
```

2. Liczba studentów obecnych na zajęciach

```
CREATE FUNCTION GetClassAttendanceCount
(
   @ClassID INT
)
RETURNS INT
AS
BEGIN
   IF EXISTS (SELECT 1 FROM Attendance WHERE ClassId = @ClassID)
       BEGIN
```

```
        DECLARE @AttendanceCount INT;


        SELECT @AttendanceCount = COUNT(*)
        FROM Attendance
        WHERE ClassID = @ClassID AND Present = 1


        RETURN @AttendanceCount;
    END
    RETURN 0
END;
```

3. Obliczanie ilości dni pozostałych do zakończenia programu edukacyjnego

```


CREATE FUNCTION DaysRemainingInProgram
(
    @ProgramID INT
)
RETURNS INT
AS
BEGIN
    DECLARE @DaysRemaining INT;


    SELECT @DaysRemaining = DATEDIFF(DAY, GETDATE(), ProgramEnd)
    FROM EducationalPrograms
    WHERE ProgramID = @ProgramID;


    -- Jeżeli program już się zakończył, zwróć 0
    IF @DaysRemaining < 0
        SET @DaysRemaining = 0;


    RETURN @DaysRemaining;
END;
```

4.Obliczanie sumy pełnym kwot za wszystkie programy na danym zamówieniu

```
CREATE FUNCTION CalculateFullPriceForOrder
(
     @OrderID int
)
RETURNS MONEY
AS
BEGIN
     DECLARE @fullprice MONEY;
```

```
    SELECT @fullprice =
    SUM(ISNULL(EP.ProgramPrice,0)) + SUM(ISNULL(C.ClassPrice,0))
    FROM Orders O
    LEFT JOIN RegisteredPrograms RP ON RP.OrderID = O.OrderID
    LEFT JOIN RegisteredClasses RC ON O.OrderID = RC.OrderID
    LEFT JOIN Classes C ON RC.ClassID = C.ClassID
    LEFT JOIN EducationalPrograms EP ON EP.ProgramID = RP.ProgramID
    GROUP BY O.OrderID
    HAVING O.OrderId = @OrderID

    RETURN ISNULL(@fullprice, 0)

END;
```

5.Obliczanie sum cen wpisowych na programy na danym zamówieniu

```
CREATE FUNCTION CalculateEntryPriceForOrder
(
    @OrderID int
)
RETURNS MONEY
AS
BEGIN
    DECLARE @entryprice MONEY;
    SELECT @entryprice =
    SUM(ISNULL(S.EntryFee,0)) + SUM(ISNULL(CS.Advance,0)) +
SUM(ISNULL(C.ClassPrice,0))
    FROM Orders O
    LEFT JOIN RegisteredPrograms RP ON RP.OrderID = O.OrderID
    LEFT JOIN RegisteredClasses RC ON O.OrderID = RC.OrderID
    LEFT JOIN Classes C ON RC.ClassID = C.ClassID
    LEFT JOIN EducationalPrograms EP ON EP.ProgramID = RP.ProgramID
    LEFT JOIN Studies S ON EP.StudiesID = S.StudiesID
    LEFT JOIN Courses CS ON CS.CourseID = EP.CourseID
    LEFT JOIN Webinars W ON W.WebinarID = EP.WebinarID
    GROUP BY O.OrderID
    HAVING O.OrderID = @OrderID

    RETURN ISNULL(@entryprice, 0)

END;
```

6.Obliczanie łącznej kwoty wydanej przez danego studenta na Programy edukacyjne

```
CREATE FUNCTION CalculateTotalPaymentsForStudent
(
   @StudentID int,
   @StartDate datetime,
   @EndDate datetime
)
RETURNS MONEY
AS
BEGIN
   DECLARE @TotalPayments MONEY;


   SELECT @TotalPayments = SUM(P.Amount)
   FROM Payments P
   WHERE P.OrderID IN (SELECT OrderID FROM Orders WHERE StudentID = @StudentID)
   AND P.Date >= @StartDate
   AND P.Date <= @EndDate
   AND status = 1;


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

5. **Triggery**

1.Aktualiacja stanu zapłaty zamówienia po udanej transakcji w tabeli Payments

```
CREATE TRIGGER trg_UpdateOrderStatus
ON Payments
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;
    IF UPDATE(Status)
    BEGIN
        IF (SELECT Status FROM INSERTED) = 1
        BEGIN
            DECLARE @OrderID INT;
            DECLARE @PaymentAmount MONEY;

            SELECT @OrderID = i.OrderID
            FROM INSERTED i;

            SELECT @PaymentAmount = i.Amount
            FROM INSERTED i;

            IF(@PaymentAmount = dbo.CalculateFullPriceForOrder(@OrderID))
            BEGIN
                UPDATE Orders
                SET OrderStatus = 'FULL PAID'
                WHERE OrderID = @OrderID
            END

            ELSE
```

```
                  BEGIN
                      IF(@PaymentAmount = dbo.CalculateFullPriceForOrder(@OrderID))
                      BEGIN
                          UPDATE Orders
                          SET OrderStatus = 'ENTRY PAID'
                          WHERE OrderID = @OrderID
                      END
                      ELSE
                      BEGIN
                          UPDATE Orders
                          SET OrderStatus = 'NOT PAID'
                          WHERE OrderID = @OrderID
                      END
                  END
              END
          END
      END;
```

6. **Indeksy**