Podstawy Baz Danych - Projekt

Autorzy: Jagoda Kurosad, Katarzyna Bęben, Oskar Blajsz

VW_StudyMeetingDurationTimeNumberOfStudents
VW_UsersPersonalData
VW_UsersDiplomasAddresses
vw_CoursesCertificatesAddresses
vw_PresenceOnPastStudyMeetings
vw_StudentsAttendanceAtSubjects
vw_bilocationReport
vw_defferedPayments
vw_MeetingsWithAbsences
VW_allUsersCourseMeetings
VW_BilocationBetweenAllActivities
VW_allUsersStudyMeetings
VW_CurrentCoursesPassed
VW_CourseModulesPassed
VW_Lecturers
VW_TranslatorWithLanguages
VW_CoursesLecturers
VW_Students
vw_StudyCoordinator
vw_WebinarTeachers
vw_CourseCoordinators
vw_InternshipCoordinators
vw_NumberOfHoursOfWOrkForAllEmployees
VW_allUsersWebinars
VW_allUsersMeetings
VW_allUsersPastStudyMeetings
VW_allUsersPastCourseMeetings
VW_allPastUsersWebinars
VW_allUsersPastMeetings
VW_allUsersFutureStudyMeetings
VW_allUsersFutureCourseMeetings
VW_allUsersFutureCourseMeetings
VW_allUsersFutureMeetings
VW_allUsersCurrentStudyMeetings
VW_allUsersCurrentCourseMeetings
VW_allUsersCurrentWebinars
VW_allUsersCurrentMeetings
VW_allUsersStationaryMeetingsWithRoomAndAddresses
VW_StudiesStartDateEndDate
VW_allOrderedActivities
VW_allFutureOrderedActivities

PROCEDURY:

AddNewStudy
AddNewCourse

AddNewWebinar
AddNewSubject
AddCourseModule
AddCourseMeeting
AddLanguage
UpdateLanguage
DeleteLanguageFromTranslatedLanguage
AddCountry
UpdateCountry
AddCity
UpdateCity
AddUser
AddEmployee
DeleteUser
DeleteWebinar
DeleteStudyMeeting
DeleteSubject
DeleteStudy
DeleteCourseMeeting
DeleteCourseModule
DeleteCourse
DeleteEmployee
UpdateUser
UpdateEmployee
UpdateCourse
UpdateWebinar
UpdateCourseModule
UpdateCourseModuleMeeting
UpdateStudies
UpdateSubject
UpdateStudyMeeting
AddUserWithRoleAndEmployee
AddTranslationLanguage
UpdateTranslatedLanguage
DeleteTranslatedLanguage
AddRoom
UpdateRoom
AddInternship
UpdateInternship
DeleteInternship
AddCourseModulePassed
UpdateCourseModulePassed
DeleteCourseModulePassed
AddUserRole

- AddStudyResult
- AddSubjectResult
- AddinternshipResult
- AddstudyMeetingPresence
- AddActivityInsteadOfPresence
- AddOrderWithDetails
- ModifyOrder
- DeleteOrder
- ModifyOrderDetail
- DeleteOrderDetail
- ModifyUserRole
- DeletedUserRole
- ModifyStudiesResult
- DeleteStudiesResult
- ModifyObjectresult
- DeleteobjectResult
- ModifyInternshipResult
- DeleteInternshipResult
- ModifyStudyMeetingPresence
- DeleteStudyMeetingPresence
- ModifyactivityInsteadOfAbsence
- Activatedeactivateuser

FUNKCJE:

- CheckIfStudentPassed
- CheckIfStudentPassedCourse
- GetStudentOrders
- GetProductsFromOrder
- CheckStudentPresenceOnActivity
- GetRemainingSeats
- GetAvailableRooms
- GetStudentAttendanceAtSubjects
- GetStudentResultsFromStudies
- GetCourseModulesPassed
- GetFutureMeetingsForStudent
- GetCurrentMeetingsForStudent
- GetNumberOfHoursOfWorkForAllEmployees
- GetUserDiplomasAndCertificates
- GetMeetingsInCity

TRIGGERY:

- trg_UpdatePaymentStatus
- trg_SetPaymentDeferred
- BeforeOrderDetailsInsert
- BeforeOrderDetailsUpdate
- PreventStudyUpdateAfterStart

- PreventSubjectUpdateAfterStart
- CheckStudyMeetingOverlap
- CheckUserRoleInsert
- CheckEmployeeExistsInUsers
- CheckTranslatedLanguageValidity
- CheckUserDeactivation

INDEXY:

- IDX_ActivitiesTypes_TypeName
- IDX_Cities_CityName_CountryID
- IDX_Countries_CountryName
- IDX_Degrees_DegreeName
- IDX_FormOfActivity_TypeName
- IDX_Grades_GradeName
- IDX_Languages_LanguageName
- IDX_Roles_RoleName
- IDX_Users_Email
- IDX_Users_CityID
- IDX_Employees_DegreeID
- IDX_Employees_EmployeeID
- IDX_Courses_CourseCoordinatorID
- IDX_Courses_CoursePrice
- IDX_Studies_StudiesCoordinatorID
- IDX_Studies_StudyPrice
- IDX_Subjects_TeacherID
- IDX_Subjects_StudiesID
- IDX_StudyMeetings_LecturerID
- IDX_StudyMeetings_TranslatorID
- IDX_StudyMeetings_SubjectID
- IDX_StudyMeetingPresence_StudentID
- IDX_StudyMeetingPresence_StudyMeetingID
- IDX_CourseModules_CourseID
- IDX_CourseModules_LecturerID
- IDX_CourseModules_TranslatorID
- IDX_CourseModulesPassed_ModuleID
- IDX_CourseModulesPassed_StudentID
- IDX_Internship_InternshipCoordinatorID
- IDX_Internship_StudiesID
- IDX_InternshipPassed_InternshipID
- IDX_InternshipPassed_StudentID
- IDX_OrderDetails_OrderID
- IDX_OrderDetails_TypeOfActivity
- IDX_Orders_StudentID
- IDX_PaymentsAdvances_DetailID
- IDX_Rooms_CityID

IDX_StationaryMeetings_RoomID
IDX_StationaryMeetings_MeetingID
IDX_StationaryCourseMeeting_ModuleID
IDX_StationaryCourseMeeting_RoomID
IDX_Webinars_TeacherID
IDX_Webinars_TranslatorID
IDX_Webinars_LanguageID
IDX_UsersRoles_UserID
IDX_UsersRoles_RoleID

ROLE:

1. Rola: admin
2. Rola: director
3. Rola: study_coordinator
4. Rola: course_coordinator
5. Rola: webinars_coordinator
6. Rola: accountant
7. Rola: secretary
8. Rola: lecturer
9. Rola: internship_coordinator
10. Rola: translator
11. Rola: student
12. Rola: guest
13. Rola: payment_system

Użytkownicy systemu

1. Administrator
2. Dyrektor Szkoły
3. Koordynator kierunku studiów
4. Koordynator kursów
5. Koordynator webinariów
6. Księgowy
7. Wykładowca
8. Osoba prowadząca praktyki
9. Tłumacz
10. Uczestnik
11. Użytkownik nieposiadający konta

12. Zewnętrzny system płatności
13. System (platforma z kursami)

W systemie obowiązuje hierarchia uprawnień użytkowników. Użytkownik będący wyżej w hierarchii rozszerza uprawnienia użytkowników:

- koordynator posiada wszystkie uprawnienia wykładowcy, tłumacza

- dyrektor posiada wszystkie uprawnienia koordynatorów
- wszyscy użytkownicy posiadają uprawnienia użytkownika nieposiadającego konta

Funkcje

1. RAPORTY

a. Raporty finansowe – zestawienie przychodów dla każdego webinaru/kursu/studium - Księgowy
b. Lista „dłużników” – osoby, które skorzystały z usług, ale nie uiściły opłat - Księgowy
c. Ogólny raport dotyczący liczby zapisanych osób na przyszłe wydarzenia (z informacją, czy wydarzenie jest stacjonarnie, czy zdalnie) - Koordynatorzy, Wykładowca (dla swoich zajęć), Dyrektor
d. Ogólny raport dotyczący frekwencji na zakończonych już wydarzeniach - Koordynatorzy, Wykładowca (dla swoich zajęć), Dyrektor
e. Lista obecności dla każdego szkolenia z datą, imieniem, nazwiskiem i informacją czy uczestnik był obecny, czy nie - Koordynatorzy, Wykładowca (dla swoich zajęć), Dyrektor
f. Raport bilokacji: lista osób, które są zapisane na co najmniej dwa przyszłe szkolenia, które ze sobą kolidują czasowo - Koordynatorzy, Dyrektor
g. Generowanie dyplomów - Koordynatorzy

2. ZARZĄDZANIE FORMĄ KSZTAŁCENIA

a. Dodawanie kursu/modułu kursu - Koordynator kursów
b. Dodawanie webinaru - Koordynator webinarów
c. Dodawanie sylabusa i harmonogramu studiów - Koordynator kierunku studiów
d. Dodawanie przedmiotów i praktyk - Koordynator kierunku studiów

e. Usuwanie formy kształcenia - Administrator

f. Modyfikacja kursu/modułu kursu - Wykładowca
g. Modyfikacja webinaru - Wykładowca
h. Modyfikacja harmonogramu studiów - Koordynator kierunku studiów
i. Modyfikacja sylabusa (kiedy jest to możliwe, czyli przed rozpoczęciem danego kierunku studiów) - Koordynator kierunku studiów

j. Umożliwienie dostępu do modułu form kształcenia bez opłaty/ z płatnością odroczoną - Dyrektor
k. Rejestrowanie obecności na zajęciach - Wykładowca/Osoba prowadząca praktyki/System

3. ZARZĄDZANIE UŻYTKOWNIKAMI BAZY

a. Dodawanie i dezaktywacja konta koordynatora formy kształcenia - Administrator

b. Dodawanie i dezaktywacja konta tłumacza - Koordynatorzy
c. Dodawanie i dezaktywacja konta wykładowcy - Koordynator kierunku studiów
d. Dodawanie i dezaktywacja konta prowadzących praktyk - Koordynator kierunku studiów
e. Dodawanie i dezaktywacja konta księgowego - Administrator
f. Dodawanie konta użytkownika - Użytkownik nieposiadający konta
g. Modyfikacja danych konta - Właściciel konkretnego konta

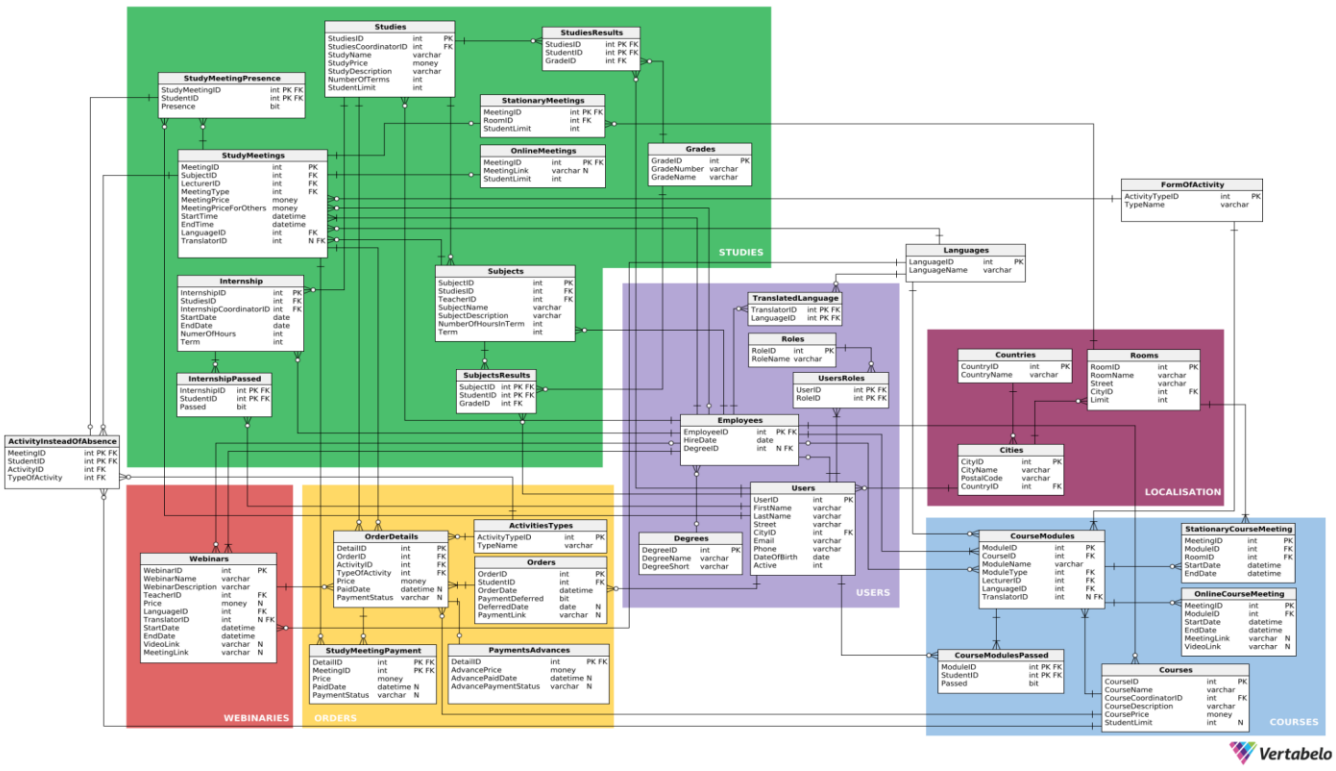
4. ZARZĄDZANIE KOSZYKIEM

a. Tworzenie zamówienia - Uczestnik
b. Dodawanie i usuwanie produktów z koszyka - Uczestnik
c. Dostęp do historii zamówień - Uczestnik
d. Generowanie linku płatności - Zewnętrzny system płatności
e. Rejestracja statusu opłaty - Zewnętrzny system płatności

5. PRZEGLĄDANIE DANYCH

a. Przeglądanie aktywności, na które jest się zapisanym - Uczestnik
b. Przeglądanie informacji o swoim przebiegu danej aktywności (np. obecność, oceny, zaliczone moduły) - Uczestnik
c. Przeglądanie dostępnych aktywności - Użytkownik nieposiadający konta
d. Przeglądanie szczegółów danej formy kształcenia (wszystkie informacje o danej formie kształcenia dostępne w bazie) - Koordynator danej formy kształcenia

Schemat bazy danych



DataBaseSchema.png

Opis tabel

ActivitiesTypes

(słownik)

OPIS

- Funkcja: Przechowuje rodzaje aktywności. Jest to tabela słownikowa.
- Kolumny:
    - ActivityTypeID *(int, NOT NULL)* - Klucz główny identyfikujący typ aktywności.
    - TypeName *(varchar(40), NOT NULL)* - Nazwa typu aktywności: studia, spotkania studyjne, kursy, webinary.

KOD

```
-- Table: ActivitiesTypes
CREATE TABLE ActivitiesTypes (
    ActivityTypeID int  NOT NULL IDENTITY(1,1),
    TypeName varchar(40)  NOT NULL,
    CONSTRAINT ActivitiesTypes_pk PRIMARY KEY  (ActivityTypeID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- ActivitiesTypes
ALTER TABLE ActivitiesTypes ADD CONSTRAINT UQ_ActivitiesTypes_TypeName
UNIQUE (TypeName);
```

ActivityInsteadOfAbsence

OPIS

- **Funkcja**: Przechowuje informacje o odrobionych przez studenta zajęciach.
- **Kolumny**:
    - MeetingID *(int, NOT NULL)* – Klucz obcy wskazujący na zajęcia z tabeli StudyMeetingPresence, które odrabia student.
    - StudentID *(int, NOT NULL)* – Klucz obcy wskazujący studenta z tabeli StudyMeetingPresence, który odrabia dane spotkanie.
    - ActivityID *(int, NOT NULL)* – Klucz obcy wskazujący na ID spotkania z tabeli StudyMeetings lub Courses, którym student odrabia swoją nieobecność.
    - TypeOfActivity *(int, NOT NULL)* – Klucz obcy wskazujący typ

aktywności z tabeli ActivitiesTypes, którym jest spotkanie, którym student odrabia swoją nieobecność.

KOD

```
-- Table: ActivityInsteadOfAbsence
CREATE TABLE ActivityInsteadOfAbsence (
    MeetingID int  NOT NULL,
    StudentID int  NOT NULL,
    ActivityID int  NOT NULL,
    TypeOfActivity int  NOT NULL,
    CONSTRAINT ActivityInsteadOfAbsence PRIMARY KEY
(MeetingID,StudentID)
);
```

KLUCZE OBCE

```
-- Reference: MeetingWithAbsence_Meeting (table:
ActivityInsteadOfAbsence)
ALTER TABLE ActivityInsteadOfAbsence ADD CONSTRAINT
MeetingAbsence_Meeting
    FOREIGN KEY (MeetingID)
    REFERENCES StudyMeetingPresence (StudyMeetingID);

-- Reference: AbsentStudent_Student (table: ActivityInsteadOfAbsence)
ALTER TABLE ActivityInsteadOfAbsence ADD CONSTRAINT
AbsentStudent_Student
    FOREIGN KEY (StudentID)
    REFERENCES StudyMeetingPresence (StudentID);

-- Reference: AbsentStudentStudiesActivity_Activity (table:
ActivityInsteadOfAbsence)
ALTER TABLE ActivityInsteadOfAbsence ADD CONSTRAINT
AbsentStudentStudiesActivity_Activity
    FOREIGN KEY (ActivityID)
    REFERENCES Courses (CourseID);

-- Reference: AbsentStudentCourseActivity_Activity (table:
ActivityInsteadOfAbsence)
ALTER TABLE ActivityInsteadOfAbsence ADD CONSTRAINT
AbsentStudentCourseActivity_Activity
    FOREIGN KEY (ActivityID)
    REFERENCES StudyMeeting (MeetingID);

-- Reference: AbsentStudentTypeOfActivity_TypeOfActivity (table:
ActivityInsteadOfAbsence)
ALTER TABLE ActivityInsteadOfAbsence ADD CONSTRAINT
AbsentStudentTypeOfActivity_TpeOfActivity
```

```
    FOREIGN KEY (TypeOfActivity)
    REFERENCES ActivitiesTypes (ActivityTypeID);
```

Cities

OPIS

- Funkcja: Przechowuje informacje o miastach z przypisaniem do krajów.
- Kolumny:
    - CityID *(int, NOT NULL)* - Klucz główny identyfikujący miasto.
    - CityName *(varchar(40), NOT NULL)* - Nazwa miasta.
    - CountryID *(int, NOT NULL)* - Klucz obcy wskazujący na tabelę Countries.

KOD

```
-- Table: Cities
CREATE TABLE Cities (
    CityID int  NOT NULL IDENTITY(1,1),
    CityName varchar(40)  NOT NULL,
    CountryID int  NOT NULL,
    CONSTRAINT Cities_pk PRIMARY KEY  (CityID)
);
```

KLUCZE OBCE

```
-- Reference: Country_City (table: Cities)
ALTER TABLE Cities ADD CONSTRAINT Country_City
    FOREIGN KEY (CountryID)
    REFERENCES Countries (CountryID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Cities
ALTER TABLE Cities ADD CONSTRAINT UQ_Cities_CityName_CountryID UNIQUE
(CityName, CountryID);
```

Countries

(słownik)

OPIS

- Funkcja: Przechowuje listę krajów. Jest to tabela słownikowa.
- Kolumny:

- CountryID *(int, NOT NULL)* - Klucz główny identyfikujący kraj.
- CountryName *(varchar(40), NOT NULL)* - Nazwa kraju.

KOD

```
-- Table: Countries
CREATE TABLE Countries (
    CountryID int  NOT NULL IDENTITY(1,1),
    CountryName varchar(40)  NOT NULL,
    CONSTRAINT Countries_pk PRIMARY KEY  (CountryID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Countries
ALTER TABLE Countries ADD CONSTRAINT UQ_Countries_CountryName UNIQUE
(CountryName);
```

CourseModules

OPIS

- Funkcja: Przechowuje informacje o modułach wchodzących w skład kursów.
- Kolumny:
  - ModuleID *(int, NOT NULL)* - Klucz główny identyfikujący moduł kursu.
  - CourseID *(int, NOT NULL)* - Klucz obcy wskazujący na tabelę Courses.
  - ModuleName *(varchar(40), NOT NULL)* - Nazwa modułu kursu.
  - ModuleType *(int, NOT NULL)* - Typ modułu (online, hybrydowy, stacjonarny).
  - LecturerID *(int, NOT NULL)* - Klucz obcy wskazujący na wykładowcę prowadzącego zajęcia.
  - LanguageID *(int, NOT NULL)* - Klucz obcy wskazujący na język zajęć z tabeli Languages.
  - TranslatorID *(int, NULL)* - Opcjonalny klucz obcy do tłumacza.

KOD

```
-- Table: CourseModules
CREATE TABLE CourseModules (
    ModuleID int  NOT NULL IDENTITY(1,1),
```

```
    CourseID int  NOT NULL,
    ModuleName varchar(40)  NOT NULL,
    ModuleType int  NOT NULL,
    LecturerID int  NOT NULL,
    LanguageID int  NOT NULL,
    TranslatorID int  NULL,
    CONSTRAINT CourseSegment_pk PRIMARY KEY  (ModuleID)
);
```

KLUCZE OBCE

```
-- Reference: CourseSegment_Courses (table: CourseModules)
ALTER TABLE CourseModules ADD CONSTRAINT CourseSegment_Courses
    FOREIGN KEY (CourseID)
    REFERENCES Courses (CourseID);

-- Reference: CourseSegment_Employees (table: CourseModules)
ALTER TABLE CourseModules ADD CONSTRAINT CourseSegment_Employees
    FOREIGN KEY (LecturerID)
    REFERENCES Employees (EmployeeID);

-- Reference: CourseSegment_Employees_1 (table: CourseModules)
ALTER TABLE CourseModules ADD CONSTRAINT CourseSegment_Employees_1
    FOREIGN KEY (TranslatorID)
    REFERENCES Employees (EmployeeID);

-- Reference: FormOfActivity_CourseModules (table: CourseModules)
ALTER TABLE CourseModules ADD CONSTRAINT FormOfActivity_CourseModules
    FOREIGN KEY (ModuleType)
    REFERENCES FormOfActivity (ActivityTypeID);

-- Reference: Languages_CourseSegment (table: CourseModules)
ALTER TABLE CourseModules ADD CONSTRAINT Languages_CourseSegment
    FOREIGN KEY (LanguageID)
    REFERENCES Languages (LanguageID);
```

CourseModulesPassed

OPIS

- Funkcja: Przechowuje informację o tym, czy student zaliczył dany moduł kursu.
- Kolumny:
    - ModuleID *(int, NOT NULL)* - Klucz obcy wskazujący na moduł kursu.

- StudentID *(int, NOT NULL)* - Klucz obcy wskazujący na studenta.
- Passed *(bit, NOT NULL)* - Status zaliczenia (1 - zaliczone, 0 - niezaliczone).

KOD

```
-- Table: CourseModulesPassed
CREATE TABLE CourseModulesPassed (
    ModuleID int  NOT NULL,
    StudentID int  NOT NULL,
    Passed bit  NOT NULL,
    CONSTRAINT CourseModulesPassed_pk PRIMARY KEY  (ModuleID,StudentID)
);
```

KLUCZE OBCE

```
-- Reference: CourseModulesPassed_CourseModules (table:
CourseModulesPassed)
ALTER TABLE CourseModulesPassed ADD CONSTRAINT
CourseModulesPassed_CourseModules
    FOREIGN KEY (ModuleID)
    REFERENCES CourseModules (ModuleID);

-- Reference: CourseModulesPassed_Users (table: CourseModulesPassed)
ALTER TABLE CourseModulesPassed ADD CONSTRAINT CourseModulesPassed_Users
    FOREIGN KEY (StudentID)
    REFERENCES Users (UserID);
```

Courses

OPIS

- Funkcja: Przechowuje informacje o kursach.
- Kolumny:
  - CourseID *(int, NOT NULL)* - Klucz główny identyfikujący kurs.
  - CourseName *(varchar(40), NOT NULL)* - Nazwa kursu.
  - CourseCoordinatorID *(int, NOT NULL)* - Klucz obcy wskazujący na koordynatora kursu.
  - CourseDescription *(varchar(255), NOT NULL)* - Opis kursu.
  - CoursePrice *(money, NOT NULL)* - Cena kursu.
  - StudentLimit *(int, NULL)* - Maksymalna liczba uczestników kursu.

KOD

```
-- Table: Courses
CREATE TABLE Courses (
    CourseID int  NOT NULL IDENTITY(1,1),
    CourseName varchar(40)  NOT NULL,
    CourseCoordinatorID int  NOT NULL,
    CourseDescription varchar(255)  NOT NULL,
    CoursePrice money  NOT NULL,
    StudentLimit int  NULL,
    CONSTRAINT Courses_pk PRIMARY KEY  (CourseID)
);
```

KLUCZE OBCE

```
-- Reference: Courses_Employees (table: Courses)
ALTER TABLE Courses ADD CONSTRAINT Courses_Employees
    FOREIGN KEY (CourseCoordinatorID)
    REFERENCES Employees (EmployeeID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Courses
ALTER TABLE Courses ADD CONSTRAINT CHK_Courses_CoursePrice CHECK
(CoursePrice >= 0);
ALTER TABLE Courses ADD CONSTRAINT CHK_Courses_NumberOfStudentsLimit
CHECK (StudentLimit > 0 OR StudentLimit IS NULL);
```

Degrees

(słownik)

OPIS

- Funkcja: Przechowuje dostępne stopnie naukowe. Jest to tabela słownikowa.
- Kolumny:
  - DegreeID *(int, NOT NULL)* - Klucz główny identyfikujący stopień naukowy.
  - DegreeName *(varchar(40), NOT NULL)* - Nazwa stopnia naukowego (np. licencjat, magister).

KOD

```
-- Table: Degrees
CREATE TABLE Degrees (
    DegreeID int  NOT NULL IDENTITY(1,1),
    DegreeName varchar(40)  NOT NULL,
```

```
    CONSTRAINT Degrees_pk PRIMARY KEY  (DegreeID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Degrees
ALTER TABLE Degrees ADD CONSTRAINT UQ_Degrees_DegreeName UNIQUE
(DegreeName);
```

Employees

OPIS

- Funkcja: Przechowuje dane o pracownikach uczelni lub organizacji.
- Kolumny:
  - EmployeeID *(int, NOT NULL)* - Klucz główny identyfikujący pracownika.
  - HireDate *(date, NOT NULL)* - Data zatrudnienia pracownika.
  - DegreeID *(int, NULL)* - Klucz obcy wskazujący na stopień naukowy z tabeli Degrees.

KOD

```
-- Table: Employees
CREATE TABLE Employees (
    EmployeeID int  NOT NULL,
    HireDate date  NOT NULL,
    DegreeID int  NULL,
    CONSTRAINT Employees_pk PRIMARY KEY  (EmployeeID)
);
```

KLUCZE OBCE

```
-- Reference: Employees_Degrees (table: Employees)
ALTER TABLE Employees ADD CONSTRAINT Employees_Degrees
    FOREIGN KEY (DegreeID)
    REFERENCES Degrees (DegreeID);

-- Reference: Employees_Users (table: Employees)
ALTER TABLE Employees ADD CONSTRAINT Employees_Users
    FOREIGN KEY (EmployeeID)
    REFERENCES Users (UserID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Employees
ALTER TABLE Employees ADD CONSTRAINT CHK_Employees_HireDate CHECK
```

```
(HireDate <= GETDATE());
```

FormOfActivity

(słownik)

OPIS

- Funkcja: Przechowuje typy formy aktywności.
- Kolumny:
  - ActivityTypeID *(int, NOT NULL)* - Klucz główny.
  - TypeName *(varchar(40), NOT NULL)* - Nazwa typu formy aktywności (online synchroniczny, online asynchroniczny, hybrydowy, stacjonarny).

KOD

```
-- Table: FormOfActivity
CREATE TABLE FormOfActivity (
    ActivityTypeID int  NOT NULL IDENTITY(1,1),
    TypeName varchar(40)  NOT NULL,
    CONSTRAINT FormOfActivity_pk PRIMARY KEY  (ActivityTypeID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- FormOfActivity
ALTER TABLE FormOfActivity ADD CONSTRAINT UQ_FormOfActivity_TypeName
UNIQUE (TypeName);
```

Grades

(słownik)

OPIS

- Funkcja: Przechowuje listę ocen.
- Kolumny:
  - GradeID *(int, NOT NULL)* - Klucz główny identyfikujący ocenę.
  - GradeName *(varchar(40), NOT NULL)* - Nazwa oceny.

KOD

```
-- Table: Grades
CREATE TABLE Grades (
    GradeID int  NOT NULL IDENTITY(1,1),
```

```
    GradeName varchar(40)  NOT NULL,
    CONSTRAINT Grades_pk PRIMARY KEY  (GradeID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Grades
ALTER TABLE Grades ADD CONSTRAINT UQ_Grades_GradeName UNIQUE
(GradeName);
```

Internship

OPIS

- Funkcja: Przechowuje dane o stażach powiązanych ze studiami.
- Kolumny:
    - InternshipID *(int, NOT NULL)* - Klucz główny.
    - StudiesID *(int, NOT NULL)* - Klucz obcy do studiów.
    - InternshipCoordinatorID *(int, NOT NULL)* - Koordynator stażu.
    - StartDate *(date, NOT NULL)* - Data rozpoczęcia stażu.
    - EndDate *(date, NOT NULL)* - Data zakończenia stażu.
    - NumerOfHours *(int, NOT NULL)* - Liczba godzin stażu.
    - Term *(int, NOT NULL)* - Semestr stażu.

KOD

```
-- Table: Internship
CREATE TABLE Internship (
    InternshipID int  NOT NULL IDENTITY(1,1),
    StudiesID int  NOT NULL,
    InternshipCoordinatorID int  NOT NULL,
    StartDate date  NOT NULL,
    EndDate date  NOT NULL,
    NumerOfHours int  NOT NULL,
    Term int  NOT NULL,
    CONSTRAINT Internship_pk PRIMARY KEY  (InternshipID)
);
```

KLUCZE OBCE

```
-- Reference: Internship_Employees (table: Internship)
ALTER TABLE Internship ADD CONSTRAINT Internship_Employees
    FOREIGN KEY (InternshipCoordinatorID)
    REFERENCES Employees (EmployeeID);
```

```
-- Reference: Internship_Studies (table: Internship)
ALTER TABLE Internship ADD CONSTRAINT Internship_Studies
    FOREIGN KEY (StudiesID)
    REFERENCES Studies (StudiesID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Internship
ALTER TABLE Internship ADD CONSTRAINT CHK_Internship_StartEndDates CHECK
(StartDate < EndDate);
ALTER TABLE Internship ADD CONSTRAINT CHK_Internship_NumerOfHours CHECK
(NumerOfHours > 0);
```

InternshipPassed

OPIS

- Funkcja: Przechowuje informację o zaliczeniu praktyk przez studenta.
- Kolumny:
    - InternshipID *(int, NOT NULL)* - Klucz obcy wskazujący praktyki.
    - StudentID *(int, NOT NULL)* - Klucz obcy wskazujący studenta.
    - Passed *(bit, NOT NULL)* - Status zaliczenia.

KOD

```
-- Table: InternshipPassed
CREATE TABLE InternshipPassed (
    InternshipID int  NOT NULL,
    StudentID int  NOT NULL,
    Passed bit  NOT NULL,
    CONSTRAINT InternshipPresence_pk PRIMARY KEY
(InternshipID,StudentID)
);
```

KLUCZE OBCE

```
-- Reference: InternshipPresence_Users (table: InternshipPassed)
ALTER TABLE InternshipPassed ADD CONSTRAINT InternshipPresence_Users
    FOREIGN KEY (StudentID)
    REFERENCES Users (UserID);

-- Reference: Internship_InternshipPresence (table: InternshipPassed)
ALTER TABLE InternshipPassed ADD CONSTRAINT
Internship_InternshipPresence
```

```
    FOREIGN KEY (InternshipID)
    REFERENCES Internship (InternshipID);
```

Languages

(słownik)

OPIS

- Funkcja: Przechowuje listę języków dostępnych na kursach lub zajęciach.
- Kolumny:
    - LanguageID *(int, NOT NULL)* - Klucz główny identyfikujący język.
    - LanguageName *(varchar(40), NOT NULL)* - Nazwa języka (np. Angielski, Polski).

KOD

```
-- Table: Languages
CREATE TABLE Languages (
    LanguageID int  NOT NULL IDENTITY(1,1),
    LanguageName varchar(40)  NOT NULL,
    CONSTRAINT Languages_pk PRIMARY KEY  (LanguageID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Languages
ALTER TABLE Languages ADD CONSTRAINT UQ_Languages_LanguageName UNIQUE
(LanguageName);
```

OnlineCourseMeeting

OPIS

- Funkcja: Przechowuje informacje o spotkaniach kursów online, takich jak daty rozpoczęcia i zakończenia oraz powiązane linki.
- Kolumny:
    - MeetingID *(int, NOT NULL)* - Klucz główny identyfikujący spotkanie.
    - ModuleID *(int, NOT NULL)* - Klucz obcy wskazujący moduł kursu.
    - StartDate *(datetime, NOT NULL)* - Data i godzina rozpoczęcia spotkania.

- EndDate *(datetime, NOT NULL)* - Data i godzina zakończenia spotkania.
- MeetingLink *(int, NULL)* - Link do spotkania online.
- VideoLink *(int, NULL)* - Link do nagrania video ze spotkania.

KOD

```
-- Table: OnlineCourseMeeting
CREATE TABLE OnlineCourseMeeting (
    MeetingID int  NOT NULL,
    ModuleID int  NOT NULL,
    StartDate datetime  NOT NULL,
    EndDate datetime  NOT NULL,
    MeetingLink int  NULL,
    VideoLink int  NULL,
    CONSTRAINT OnlineCourseMeeting_pk PRIMARY KEY  (MeetingID)
);
```

KLUCZE OBCE

```
-- Reference: OnlineModules_CourseModules (table: OnlineCourseMeeting)
ALTER TABLE OnlineCourseMeeting ADD CONSTRAINT
OnlineModules_CourseModules
    FOREIGN KEY (ModuleID)
    REFERENCES CourseModules (ModuleID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- OnlineCourseMeeting
ALTER TABLE OnlineCourseMeeting ADD CONSTRAINT
CHK_OnlineCourseMeeting_StartEndDates CHECK (StartDate < EndDate);
```

OnlineMeetings

OPIS

- Funkcja: Przechowuje dane o ogólnych spotkaniach online z limitem uczestników.
- Kolumny:
  - MeetingID *(int, NOT NULL)* - Klucz główny identyfikujący spotkanie.
  - MeetingLink *(int, NULL)* - Link do spotkania online.
  - StudentLimit *(int, NOT NULL)* - Maksymalna liczba uczestników.

KOD

```
-- Table: OnlineMeetings
CREATE TABLE OnlineMeetings (
    MeetingID int  NOT NULL,
    MeetingLink int  NULL,
    StudentLimit int  NOT NULL,
    CONSTRAINT OnlineMeetings_pk PRIMARY KEY  (MeetingID)
);
```

KLUCZE OBCE

```
-- Reference: OnlineMeetings_StudyMeetings (table: OnlineMeetings)
ALTER TABLE OnlineMeetings ADD CONSTRAINT OnlineMeetings_StudyMeetings
    FOREIGN KEY (MeetingID)
    REFERENCES StudyMeetings (MeetingID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- OnlineMeetings
ALTER TABLE OnlineMeetings ADD CONSTRAINT
CHK_OnlineMeetings_StudentLimit CHECK (StudentLimit > 0 OR StudentLimit
IS NULL);
```

OrderDetails

OPIS

- Funkcja: Przechowuje szczegóły zamówienia, w tym powiązanie z aktywnościami i cenami.
- Kolumny:
    - `DetailID` *(int, NOT NULL)* - Klucz główny identyfikujący szczegół zamówienia.
    - `OrderID` *(int, NOT NULL)* - Klucz obcy wskazujący zamówienie z tabeli `Orders`.
    - `ActivityID` *(int, NOT NULL)* - Identyfikator aktywności powiązanej z zamówieniem.
    - `TypeOfActivity` *(int, NOT NULL)* - Typ aktywności.
    - `Price` *(money, NOT NULL)* - Cena za aktywność.
    - `PaidDate` *(datetime, NULL)* - Data opłacenia.
    - `PaymentStatus` *(varchar(40), NULL)* - Status płatności (udana, nieudana).

KOD

```
-- Table: OrderDetails
CREATE TABLE OrderDetails (
```

```
    DetailID int  NOT NULL IDENTITY(1,1),
    OrderID int  NOT NULL,
    ActivityID int  NOT NULL,
    TypeOfActivity int  NOT NULL,
    Price money  NOT NULL,
    PaidDate datetime  NULL,
    PaymentStatus varchar(40)  NULL,
    CONSTRAINT OrderDetails_pk PRIMARY KEY  (DetailID)
);
```

KLUCZE OBCE

```
-- Reference: OrderDetails_ActivitiesTypes (table: OrderDetails)
ALTER TABLE OrderDetails ADD CONSTRAINT OrderDetails_ActivitiesTypes
    FOREIGN KEY (TypeOfActivity)
    REFERENCES ActivitiesTypes (ActivityTypeID);

-- Reference: OrderDetails_Courses (table: OrderDetails)
ALTER TABLE OrderDetails ADD CONSTRAINT OrderDetails_Courses
    FOREIGN KEY (ActivityID)
    REFERENCES Courses (CourseID);

-- Reference: OrderDetails_Studies (table: OrderDetails)
ALTER TABLE OrderDetails ADD CONSTRAINT OrderDetails_Studies
    FOREIGN KEY (ActivityID)
    REFERENCES Studies (StudiesID);

-- Reference: OrderDetails_StudyMeetings (table: OrderDetails)
ALTER TABLE OrderDetails ADD CONSTRAINT OrderDetails_StudyMeetings
    FOREIGN KEY (ActivityID)
    REFERENCES StudyMeetings (MeetingID);

-- Reference: OrderDetails_Webinars (table: OrderDetails)
ALTER TABLE OrderDetails ADD CONSTRAINT OrderDetails_Webinars
    FOREIGN KEY (ActivityID)
    REFERENCES Webinars (WebinarID);

-- Reference: Orders_OrderDetails (table: OrderDetails)
ALTER TABLE OrderDetails ADD CONSTRAINT Orders_OrderDetails
    FOREIGN KEY (OrderID)
    REFERENCES Orders (OrderID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- OrderDetails
ALTER TABLE OrderDetails ADD CONSTRAINT CHK_OrderDetails_PaymentStatus
CHECK (PaymentStatus in ('udana', 'nieudana'));
```

```
-- OrderDetails
ALTER TABLE OrderDetails ADD CONSTRAINT CHK_OrderDetails_Price CHECK
(Price >= 0);
```

Orders

OPIS

- Funkcja: Przechowuje dane o zamówieniach dokonanych przez studentów.
- Kolumny:
  - OrderID *(int, NOT NULL)* - Klucz główny identyfikujący zamówienie.
  - StudentID *(int, NOT NULL)* - Klucz obcy wskazujący studenta.
  - OrderDate *(datetime, NOT NULL)* - Data zamówienia.
  - PaymentDeferred *(bit, NOT NULL)* - Czy płatność została odroczona (1 - tak, 0 - nie).
  - DeferredDate *(date, NULL)* - Data odroczonej płatności.
  - PaymentLink *(varchar(255), NULL)* - Link do płatności online.

KOD

```
-- Table: Orders
CREATE TABLE Orders (
    OrderID int  NOT NULL IDENTITY(1,1),
    StudentID int  NOT NULL,
    OrderDate datetime  NOT NULL,
    PaymentDeferred bit  NOT NULL,
    DeferredDate date  NULL,
    PaymentLink varchar(255)  NULL,
    CONSTRAINT Orders_pk PRIMARY KEY  (OrderID)
);
```

KLUCZE OBCE

```
-- Reference: Users_Orders (table: Orders)
ALTER TABLE Orders ADD CONSTRAINT Users_Orders
    FOREIGN KEY (StudentID)
    REFERENCES Users (UserID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Orders
ALTER TABLE Orders ADD CONSTRAINT CHK_Orders_DeferredDate CHECK
(DeferredDate > OrderDate)
```

PaymentsAdvances

OPIS

- Funkcja: Przechowuje informacje o zaliczkach wpłaconych w ramach zamówień.
- Kolumny:
  - DetailID *(int, NOT NULL)* - Klucz główny, powiązany z tabelą OrderDetails.
  - AdvancePrice *(money, NOT NULL)* - Wysokość zaliczki.
  - AdvancePaidDate *(datetime, NULL)* - Data wpłaty zaliczki.
  - AdvancePaymentStatus *(varchar(40), NULL)* - Status płatności zaliczki.

KOD

```
-- Table: PaymentsAdvances
CREATE TABLE PaymentsAdvances (
    DetailID int  NOT NULL,
    AdvancePrice money  NOT NULL,
    AdvancePaidDate datetime  NULL,
    AdvancePaymentStatus varchar(40)  NULL,
    CONSTRAINT PaymentsAdvances_pk PRIMARY KEY  (DetailID)
);
```

KLUCZE OBCE

```
-- Reference: PaymentsAdvances_OrderDetails (table: PaymentsAdvances)
ALTER TABLE PaymentsAdvances ADD CONSTRAINT
PaymentsAdvances_OrderDetails
    FOREIGN KEY (DetailID)
    REFERENCES OrderDetails (DetailID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- PaymentsAdvances
ALTER TABLE PaymentsAdvances
ADD CONSTRAINT CHK_PaymentsAdvances_AdvancePrice CHECK (AdvancePrice >=
0);

ALTER TABLE PaymentsAdvances ADD CONSTRAINT
CHK_PaymentsAdvances_AdvancePaymentStatus CHECK
(AdvancePaymentStatus in ('udana', 'nieudana'))
```

Roles

(słownik)

OPIS

- Funkcja: Przechowuje role użytkowników w systemie.
- Kolumny:
  - RoleID *(int, NOT NULL)* - Klucz główny identyfikujący rolę.
  - RoleName *(varchar(40), NOT NULL)* - Nazwa roli (np. Student, Wykładowca).

KOD

```
-- Table: Roles
CREATE TABLE Roles (
    RoleID int  NOT NULL IDENTITY(1,1),
    RoleName varchar(40)  NOT NULL,
    CONSTRAINT Roles_pk PRIMARY KEY  (RoleID)
);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Roles
ALTER TABLE Roles ADD CONSTRAINT UQ_Roles_RoleName UNIQUE (RoleName);
```

Rooms

OPIS

- Funkcja: Przechowuje informacje o salach przeznaczonych na zajęcia stacjonarne.
- Kolumny:
  - RoomID *(int, NOT NULL)* - Klucz główny identyfikujący salę.
  - RoomName *(varchar(40), NOT NULL)* - Nazwa lub numer sali.
  - Street *(varchar(40), NOT NULL)* - Ulica, na której znajduje się sala.
  - PostalCode *(varchar(6), NOT NULL)* - Kod pocztowy.
  - CityID *(int, NOT NULL)* - Klucz obcy wskazujący miasto.
  - Limit *(int, NOT NULL)* - Maksymalna liczba uczestników zajęć.

KOD

```
-- Table: Rooms
```

```
CREATE TABLE Rooms (
    RoomID int  NOT NULL IDENTITY(1,1),
    RoomName varchar(40)  NOT NULL,
    Street varchar(40)  NOT NULL,
    PostalCode varchar(6)  NOT NULL,
    CityID int  NOT NULL,
    Limit int  NOT NULL,
    CONSTRAINT Rooms_pk PRIMARY KEY  (RoomID)
);
```

KLUCZE OBCE

```
-- Reference: Rooms_Cities (table: Rooms)
ALTER TABLE Rooms ADD CONSTRAINT Rooms_Cities
    FOREIGN KEY (CityID)
    REFERENCES Cities (CityID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Rooms
ALTER TABLE Rooms ADD CONSTRAINT CHK_Rooms_Limit CHECK (Limit > 0);
```

StationaryCourseMeeting

OPIS

- Funkcja: Przechowuje szczegóły spotkań kursów stacjonarnych.
- Kolumny:
  - MeetingID *(int, NOT NULL)* - Klucz główny identyfikujący spotkanie.
  - ModuleID *(int, NOT NULL)* - Klucz obcy do modułu kursu.
  - RoomID *(int, NOT NULL)* - Klucz obcy wskazujący salę z tabeli Rooms.
  - StartDate *(datetime, NOT NULL)* - Data i godzina rozpoczęcia zajęć.
  - EndDate *(datetime, NOT NULL)* - Data i godzina zakończenia zajęć.

KOD

```
-- Table: StationaryCourseMeeting
CREATE TABLE StationaryCourseMeeting (
    MeetingID int  NOT NULL,
    ModuleID int  NOT NULL,
    RoomID int  NOT NULL,
    StartDate datetime  NOT NULL,
```

```
    EndDate datetime  NOT NULL,
    CONSTRAINT StationaryCourseMeeting_pk PRIMARY KEY  (MeetingID)
);
```

KLUCZE OBCE

```
-- Reference: StationaryModules_CourseModules (table:
StationaryCourseMeeting)
ALTER TABLE StationaryCourseMeeting ADD CONSTRAINT
StationaryModules_CourseModules
    FOREIGN KEY (ModuleID)
    REFERENCES CourseModules (ModuleID);

-- Reference: StationaryModules_Rooms (table: StationaryCourseMeeting)
ALTER TABLE StationaryCourseMeeting ADD CONSTRAINT
StationaryModules_Rooms
    FOREIGN KEY (RoomID)
    REFERENCES Rooms (RoomID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- StationaryCourseMeeting
ALTER TABLE StationaryCourseMeeting ADD CONSTRAINT
CHK_StationaryCourseMeeting_StartEndDates CHECK (StartDate < EndDate);
```

StationaryMeetings

OPIS

- Funkcja: Przechowuje dane dotyczące ogólnych spotkań stacjonarnych z limitem uczestników.
- Kolumny:
    - MeetingID *(int, NOT NULL)* - Klucz główny identyfikujący spotkanie.
    - RoomID *(int, NOT NULL)* - Klucz obcy wskazujący salę.
    - StudentLimit *(int, NOT NULL)* - Maksymalna liczba studentów uczestniczących w spotkaniu.

KOD

```
-- Table: StationaryMeetings
CREATE TABLE StationaryMeetings (
    MeetingID int  NOT NULL,
    RoomID int  NOT NULL,
    StudentLimit int  NOT NULL,
    CONSTRAINT StationaryMeetings_pk PRIMARY KEY  (MeetingID)
```

```
);
```

KLUCZE OBCE

```
-- Reference: StationaryMeetings_Rooms (table: StationaryMeetings)
ALTER TABLE StationaryMeetings ADD CONSTRAINT StationaryMeetings_Rooms
    FOREIGN KEY (RoomID)
    REFERENCES Rooms (RoomID);

-- Reference: StationaryMeetings_StudyMeetings (table:
StationaryMeetings)
ALTER TABLE StationaryMeetings ADD CONSTRAINT
StationaryMeetings_StudyMeetings
    FOREIGN KEY (MeetingID)
    REFERENCES StudyMeetings (MeetingID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- StationaryMeetings
ALTER TABLE StationaryMeetings ADD CONSTRAINT
CHK_StationaryMeetings_StudentLimit CHECK (StudentLimit > 0);
```

Studies

OPIS

- Funkcja: Przechowuje dane o kierunkach studiów oferowanych przez uczelnię.
- Kolumny:
  - StudiesID *(int, NOT NULL)* - Klucz główny identyfikujący kierunek studiów.
  - StudiesCoordinatorID *(int, NOT NULL)* - Koordynator studiów.
  - StudyName *(varchar(40), NOT NULL)* - Nazwa kierunku studiów.
  - StudyPrice *(money, NOT NULL)* - Cena wpisowego.
  - StudyDescription *(varchar(255), NOT NULL)* - Opis kierunku studiów.
  - NumberOfTerms *(int, NOT NULL)* - Liczba semestrów.
  - StudentLimit *(int, NOT NULL)* - Maksymalna liczba studentów na kierunku.

KOD

```
-- Table: Studies
CREATE TABLE Studies (
    StudiesID int  NOT NULL IDENTITY(1,1),
    StudiesCoordinatorID int  NOT NULL,
```

```
    StudyName varchar(40)  NOT NULL,
    StudyPrice money  NOT NULL,
    StudyDescription varchar(255)  NOT NULL,
    NumberOfTerms int  NOT NULL,
    StudentLimit int  NOT NULL,
    CONSTRAINT Studies_pk PRIMARY KEY  (StudiesID)
);
```

KLUCZE OBCE

```
-- Reference: Studies_Employees (table: Studies)
ALTER TABLE Studies ADD CONSTRAINT Studies_Employees
    FOREIGN KEY (StudiesCoordinatorID)
    REFERENCES Employees (EmployeeID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Studies
ALTER TABLE Studies ADD CONSTRAINT CHK_Studies_StudyPrice CHECK
(StudyPrice >= 0);
ALTER TABLE Studies ADD CONSTRAINT CHK_Studies_NumberOfTerms CHECK
(NumberOfTerms > 0);
```

StudiesResults

OPIS

- **Funkcja**: Przechowuje wyniki studentów z poszczególnych kierunków studiów.
- **Kolumny**:
    - StudiesID *(int, NOT NULL)* – Klucz obcy wskazujący na kierunek studiów z tabeli Studies.
    - StudentID *(int, NOT NULL)* – Klucz obcy wskazujący na studenta z tabeli Users.
    - GradeID *(int, NOT NULL)* – Klucz obcy wskazujący ocenę z tabeli Grades.

KOD

```
-- Table: StudiesResults
CREATE TABLE StudiesResults (
    StudiesID int  NOT NULL,
    StudentID int  NOT NULL,
    GradeID int  NOT NULL,
    CONSTRAINT StudiesResults_pk PRIMARY KEY  (StudiesID,StudentID)
```

```
);
```

KLUCZE OBCE

```
-- Reference: StudiesResults_Grades (table: StudiesResults)
ALTER TABLE StudiesResults ADD CONSTRAINT StudiesResults_Grades
    FOREIGN KEY (GradeID)
    REFERENCES Grades (GradeID);

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

-- Reference: Users_StudiesResults (table: StudiesResults)
ALTER TABLE StudiesResults ADD CONSTRAINT Users_StudiesResults
    FOREIGN KEY (StudentID)
    REFERENCES Users (UserID);
```

StudyMeetingPayment

OPIS

- **Funkcja**: Przechowuje informacje o płatnościach za uczestnictwo w spotkaniach studenckich.
- **Kolumny**:
  - `DetailID` *(int, NOT NULL)* – Klucz główny identyfikujący szczegół płatności.
  - `MeetingID` *(int, NOT NULL)* – Klucz obcy wskazujący na spotkanie z tabeli `StudyMeetings`.
  - `Price` *(money, NOT NULL)* – Kwota płatności.
  - `PaidDate` *(datetime, NULL)* – Data dokonania płatności.
  - `PaymentStatus` *(varchar(40), NULL)* – Status płatności (udana, nieudana).

KOD

```
-- Table: StudyMeetingPayment
CREATE TABLE StudyMeetingPayment (
    DetailID int  NOT NULL,
    MeetingID int  NOT NULL,
    Price money  NOT NULL,
    PaidDate datetime  NULL,
    PaymentStatus varchar(40)  NULL,
```

```
    CONSTRAINT StudyMeetingPayment_pk PRIMARY KEY  (DetailID)
);
```

KLUCZE OBCE

```
-- Reference: OrderDetails_StudyMeetingPayment (table:
StudyMeetingPayment)
ALTER TABLE StudyMeetingPayment ADD CONSTRAINT
OrderDetails_StudyMeetingPayment
    FOREIGN KEY (DetailID)
    REFERENCES OrderDetails (DetailID);

-- Reference: StudyMeetings_StudyMeetingPayment (table:
StudyMeetingPayment)
ALTER TABLE StudyMeetingPayment ADD CONSTRAINT
StudyMeetings_StudyMeetingPayment
    FOREIGN KEY (MeetingID)
    REFERENCES StudyMeetings (MeetingID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- StudyMeetingPayment
ALTER TABLE StudyMeetingPayment ADD CONSTRAINT
CHK_StudyMeetingPayment_PaymentStatus CHECK (PaymentStatus in
('udana', 'nieudana'));

ALTER TABLE StudyMeetingPayment ADD CONSTRAINT
CHK_StudyMeetingPayment_Price CHECK (Price > 0);
```

StudyMeetingPresence

OPIS

- **Funkcja**: Przechowuje informacje o obecności studentów na spotkaniach studenckich.
- **Kolumny**:
    - StudyMeetingID *(int, NOT NULL)* – Klucz obcy wskazujący na spotkanie z tabeli StudyMeetings.
    - StudentID *(int, NOT NULL)* – Klucz obcy wskazujący na studenta z tabeli Users.
    - Presence *(bit, NOT NULL)* – Informacja o obecności (1 – obecny, 0 – nieobecny).

KOD

```
-- Table: StudyMeetingPresence
```

```sql
CREATE TABLE StudyMeetingPresence (
    StudyMeetingID int  NOT NULL,
    StudentID int  NOT NULL,
    Presence bit  NOT NULL,
    CONSTRAINT StudyMeetingPresence_pk PRIMARY KEY
(StudyMeetingID,StudentID)
);
```

KLUCZE OBCE

```sql
-- Reference: StudyMeetingPresence_Users (table: StudyMeetingPresence)
ALTER TABLE StudyMeetingPresence ADD CONSTRAINT
StudyMeetingPresence_Users
    FOREIGN KEY (StudentID)
    REFERENCES Users (UserID);

-- Reference: StudyMeetings_Presence (table: StudyMeetingPresence)
ALTER TABLE StudyMeetingPresence ADD CONSTRAINT StudyMeetings_Presence
    FOREIGN KEY (StudyMeetingID)
    REFERENCES StudyMeetings (MeetingID);
```

StudyMeetings

OPIS

- **Funkcja**: Przechowuje szczegółowe dane dotyczące spotkań studenckich, w tym opłaty, daty i język spotkania.
- **Kolumny**:
    - `MeetingID` *(int, NOT NULL)* – Klucz główny identyfikujący spotkanie.
    - `SubjectID` *(int, NOT NULL)* – Klucz obcy wskazujący na przedmiot z tabeli `Subjects`.
    - `LecturerID` *(int, NOT NULL)* – Klucz obcy wskazujący na prowadzącego z tabeli `Employees`.
    - `MeetingType` *(int, NOT NULL)* – Typ spotkania (online, stacjonarne).
    - `MeetingPrice` *(money, NOT NULL)* – Cena za uczestnictwo w spotkaniu.
    - `MeetingPriceForOthers` *(money, NOT NULL)* – Cena dla uczestników zewnętrznych.
    - `StartTime` *(datetime, NOT NULL)* – Data i godzina rozpoczęcia spotkania.
    - `EndTime` *(datetime, NOT NULL)* – Data i godzina zakończenia spotkania.
    - `LanguageID` *(int, NOT NULL)* – Klucz obcy wskazujący język

spotkania z tabeli Languages.

- TranslatorID *(int, NULL)* – Klucz obcy wskazujący tłumacza z tabeli Employees.

KOD

```
-- Table: StudyMeetings
CREATE TABLE StudyMeetings (
    MeetingID int  NOT NULL IDENTITY(1,1),
    SubjectID int  NOT NULL,
    LecturerID int  NOT NULL,
    MeetingType int  NOT NULL,
    MeetingPrice money  NOT NULL,
    MeetingPriceForOthers money  NOT NULL,
    StartTime datetime  NOT NULL,
    EndTime datetime  NOT NULL,
    LanguageID int  NOT NULL,
    TranslatorID int  NULL,
    CONSTRAINT StudyMeetings_pk PRIMARY KEY  (MeetingID)
);
```

KLUCZE OBCE

```
-- Reference: FormOfActivity_StudyMeetings (table: StudyMeetings)
ALTER TABLE StudyMeetings ADD CONSTRAINT FormOfActivity_StudyMeetings
    FOREIGN KEY (MeetingType)
    REFERENCES FormOfActivity (ActivityTypeID);

-- Reference: StudyMeetings_Employees (table: StudyMeetings)
ALTER TABLE StudyMeetings ADD CONSTRAINT StudyMeetings_Employees
    FOREIGN KEY (TranslatorID)
    REFERENCES Employees (EmployeeID);

-- Reference: StudyMeetings_Employees_00 (table: StudyMeetings)
ALTER TABLE StudyMeetings ADD CONSTRAINT StudyMeetings_Employees_00
    FOREIGN KEY (LecturerID)
    REFERENCES Employees (EmployeeID);

-- Reference: StudyMeetings_Languages (table: StudyMeetings)
ALTER TABLE StudyMeetings ADD CONSTRAINT StudyMeetings_Languages
    FOREIGN KEY (LanguageID)
    REFERENCES Languages (LanguageID);

-- Reference: StudyMeetings_Subjects (table: StudyMeetings)
ALTER TABLE StudyMeetings ADD CONSTRAINT StudyMeetings_Subjects
    FOREIGN KEY (SubjectID)
    REFERENCES Subjects (SubjectID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- StudyMeetings
ALTER TABLE StudyMeetings ADD CONSTRAINT CHK_StudyMeetings_StartEndTimes
CHECK (StartTime < EndTime);
ALTER TABLE StudyMeetings ADD CONSTRAINT CHK_StudyMeetings_Price CHECK
(MeetingPrice >= 0);
ALTER TABLE StudyMeetings ADD CONSTRAINT
CHK_StudyMeetings_PriceForOthers CHECK (MeetingPriceForOthers >= 0);
```

Subjects

OPIS

- **Funkcja**: Przechowuje informacje o przedmiotach związanych z poszczególnymi kierunkami studiów.
- **Kolumny**:
  - SubjectID *(int, NOT NULL)* – Klucz główny identyfikujący przedmiot.
  - StudiesID *(int, NOT NULL)* – Klucz obcy wskazujący na kierunek studiów z tabeli Studies.
  - TeacherID *(int, NOT NULL)* – Klucz obcy wskazujący prowadzącego przedmiot z tabeli Employees.
  - SubjectName *(varchar(40), NOT NULL)* – Nazwa przedmiotu.
  - SubjectDescription *(varchar(255), NOT NULL)* – Opis przedmiotu.
  - NumberOfHoursInTerm *(int, NOT NULL)* – Liczba godzin zajęć w jednym semestrze.
  - Term *(int, NOT NULL)* – Semestr, w którym przedmiot jest realizowany.

KOD

```
-- Table: Subjects
CREATE TABLE Subjects (
    SubjectID int  NOT NULL IDENTITY(1,1),
    StudiesID int  NOT NULL,
    TeacherID int  NOT NULL,
    SubjectName varchar(40)  NOT NULL,
    SubjectDescription varchar(255)  NOT NULL,
    NumberOfHoursInTerm int  NOT NULL,
    Term int  NOT NULL,
    CONSTRAINT Subjects_pk PRIMARY KEY  (SubjectID)
);
```

KLUCZE OBCE

```
-- Reference: Subjects_Employees (table: Subjects)
ALTER TABLE Subjects ADD CONSTRAINT Subjects_Employees
    FOREIGN KEY (TeacherID)
    REFERENCES Employees (EmployeeID);

-- Reference: Subjects_Studies (table: Subjects)
ALTER TABLE Subjects ADD CONSTRAINT Subjects_Studies
    FOREIGN KEY (StudiesID)
    REFERENCES Studies (StudiesID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Subjects
ALTER TABLE Subjects ADD CONSTRAINT CHK_Subjects_NumberOfHoursInTerm
CHECK (NumberOfHoursInTerm > 0);
```

SubjectsResults

OPIS

- **Funkcja**: Przechowuje wyniki studentów z poszczególnych przedmiotów.
- **Kolumny**:
  - SubjectID *(int, NOT NULL)* – Klucz obcy wskazujący przedmiot z tabeli Subjects.
  - StudentID *(int, NOT NULL)* – Klucz obcy wskazujący studenta z tabeli Users.
  - GradeID *(int, NOT NULL)* – Klucz obcy wskazujący ocenę z tabeli Grades.

KOD

```
-- Table: SubjectsResults
CREATE TABLE SubjectsResults (
    SubjectID int  NOT NULL,
    StudentID int  NOT NULL,
    GradeID int  NOT NULL,
    CONSTRAINT SubjectsResults_pk PRIMARY KEY  (SubjectID,StudentID)
);
```

KLUCZE OBCE

```
-- Reference: SubjectsResults_Grades (table: SubjectsResults)
ALTER TABLE SubjectsResults ADD CONSTRAINT SubjectsResults_Grades
    FOREIGN KEY (GradeID)
```

```
    REFERENCES Grades (GradeID);

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

-- Reference: Users_SubjectsResults (table: SubjectsResults)
ALTER TABLE SubjectsResults ADD CONSTRAINT Users_SubjectsResults
    FOREIGN KEY (StudentID)
    REFERENCES Users (UserID);
```

TranslatedLanguage

OPIS

- **Funkcja**: Przechowuje powiązania między tłumaczami a językami, które obsługują.
- **Kolumny**:
    - TranslatorID *(int, NOT NULL)* – Klucz obcy wskazujący na tłumacza z tabeli Employees.
    - LanguageID *(int, NOT NULL)* – Klucz obcy wskazujący język z tabeli Languages.

KOD

```
-- Table: TranslatedLanguage
CREATE TABLE TranslatedLanguage (
    TranslatorID int  NOT NULL,
    LanguageID int  NOT NULL,
    CONSTRAINT TranslatedLanguage_pk PRIMARY KEY
(LanguageID,TranslatorID)
);
```

KLUCZE OBCE

```
-- Reference: TranslatedLanguage_Employees (table: TranslatedLanguage)
ALTER TABLE TranslatedLanguage ADD CONSTRAINT
TranslatedLanguage_Employees
    FOREIGN KEY (TranslatorID)
    REFERENCES Employees (EmployeeID);

-- Reference: TranslatedLanguage_Languages (table: TranslatedLanguage)
ALTER TABLE TranslatedLanguage ADD CONSTRAINT
TranslatedLanguage_Languages
    FOREIGN KEY (LanguageID)
```

```
    REFERENCES Languages (LanguageID);
```

Users

OPIS

- **Funkcja**: Przechowuje dane użytkowników systemu, w tym studentów i pracowników.
- **Kolumny**:
  - UserID *(int, NOT NULL)* – Klucz główny identyfikujący użytkownika.
  - FirstName *(varchar(40), NOT NULL)* – Imię użytkownika.
  - LastName *(varchar(40), NOT NULL)* – Nazwisko użytkownika.
  - Street *(varchar(40), NOT NULL)* – Adres użytkownika – ulica.
  - PostalCode *(varchar(6), NOT NULL)* – Kod pocztowy.
  - CityID *(int, NOT NULL)* – Klucz obcy wskazujący na miasto z tabeli Cities.
  - Email *(varchar(40), NOT NULL)* – Adres e-mail użytkownika. *(warunek regex)*
  - Phone *(varchar(40), NOT NULL)* – Numer telefonu użytkownika.
  - DateOfBirth *(date, NOT NULL)* – Data urodzenia użytkownika.

KOD

```
-- Table: Users
CREATE TABLE Users (
    UserID int  NOT NULL IDENTITY(1,1),
    FirstName varchar(40)  NOT NULL,
    LastName varchar(40)  NOT NULL,
    Street varchar(40)  NOT NULL,
    PostalCode varchar(6)  NOT NULL,
    CityID int  NOT NULL,
    Email varchar(40)  NOT NULL,
    Phone varchar(40)  NOT NULL,
    DateOfBirth date  NOT NULL,
    CONSTRAINT Users_pk PRIMARY KEY  (UserID)
);
```

KLUCZE OBCE

```
-- Reference: City_User (table: Users)
ALTER TABLE Users ADD CONSTRAINT City_User
    FOREIGN KEY (CityID)
    REFERENCES Cities (CityID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Users
ALTER TABLE Users ADD CONSTRAINT CHK_Users_DateOfBirth CHECK
(DateOfBirth < GETDATE());
ALTER TABLE Users ADD CONSTRAINT CHK_Users_Email_LIKE CHECK (Email LIKE
'%@%.%');
ALTER TABLE Users ADD CONSTRAINT UQ_Users_Email UNIQUE (Email);
```

UsersRoles

OPIS

- **Funkcja**: Przechowuje przypisanie ról do użytkowników systemu.
- **Kolumny**:
  - `UserID` *(int, NOT NULL)* – Klucz obcy wskazujący na użytkownika z tabeli `Users`.
  - `RoleID` *(int, NOT NULL)* – Klucz obcy wskazujący rolę z tabeli `Roles`.

KOD

```
-- Table: UsersRoles
CREATE TABLE UsersRoles (
    UserID int  NOT NULL,
    RoleID int  NOT NULL,
    CONSTRAINT UsersRoles_pk PRIMARY KEY  (UserID,RoleID)
);
```

KLUCZE OBCE

```
-- Reference: EmployeeRoles_Roles (table: UsersRoles)
ALTER TABLE UsersRoles ADD CONSTRAINT EmployeeRoles_Roles
    FOREIGN KEY (RoleID)
    REFERENCES Roles (RoleID);

-- Reference: UsersRoles_Users (table: UsersRoles)
ALTER TABLE UsersRoles ADD CONSTRAINT UsersRoles_Users
    FOREIGN KEY (UserID)
    REFERENCES Users (UserID);
```

Webinars

OPIS

- **Funkcja**: Przechowuje informacje o webinarach, w tym linki, język prowadzenia i tłumacza.
- **Kolumny**:
  - WebinarID *(int, NOT NULL)* – Klucz główny identyfikujący webinar.
  - WebinarName *(varchar(40), NOT NULL)* – Nazwa webinaru.
  - WebinarDescription *(varchar(255), NOT NULL)* – Opis webinaru.
  - TeacherID *(int, NOT NULL)* – Klucz obcy wskazujący prowadzącego webinar z tabeli Employees.
  - Price *(money, NULL)* – Cena webinaru.
  - LanguageID *(int, NOT NULL)* – Klucz obcy wskazujący język webinaru z tabeli Languages.
  - TranslatorID *(int, NULL)* – Klucz obcy wskazujący tłumacza webinaru.
  - StartDate *(datetime, NOT NULL)* – Data i godzina rozpoczęcia webinaru.
  - EndDate *(datetime, NOT NULL)* – Data i godzina zakończenia webinaru.
  - VideoLink *(varchar(255), NULL)* – Link do nagrania video webinaru.
  - MeetingLink *(varchar(255), NULL)* – Link do spotkania online.

KOD

```
-- Table: Webinars
CREATE TABLE Webinars (
    WebinarID int  NOT NULL IDENTITY(1,1),
    WebinarName varchar(40)  NOT NULL,
    WebinarDescription varchar(255)  NOT NULL,
    TeacherID int  NOT NULL,
    Price money  NULL,
    LanguageID int  NOT NULL,
    TranslatorID int  NULL,
    StartDate datetime  NOT NULL,
    EndDate datetime  NOT NULL,
    VideoLink varchar(255)  NULL,
    MeetingLink varchar(255)  NULL,
    CONSTRAINT Webinars_pk PRIMARY KEY  (WebinarID)
);
```

KLUCZE OBCE

```
-- Reference: Webinars_Employees (table: Webinars)
ALTER TABLE Webinars ADD CONSTRAINT Webinars_Employees
    FOREIGN KEY (TranslatorID)
    REFERENCES Employees (EmployeeID);

-- Reference: Webinars_Employees_01 (table: Webinars)
ALTER TABLE Webinars ADD CONSTRAINT Webinars_Employees_01
    FOREIGN KEY (TeacherID)
    REFERENCES Employees (EmployeeID);

-- Reference: Webinars_Languages (table: Webinars)
ALTER TABLE Webinars ADD CONSTRAINT Webinars_Languages
    FOREIGN KEY (LanguageID)
    REFERENCES Languages (LanguageID);
```

WARUNKI INTEGRALNOŚCIOWE

```
-- Webinars
ALTER TABLE Webinars ADD CONSTRAINT CHK_Webinars_StartEndDates CHECK
(StartDate < EndDate);
ALTER TABLE Webinars ADD CONSTRAINT CHK_Webinars_Price CHECK (Price >= 0);
```

Generowanie danych

Dane generowaliśmy za pomocą skryptów napisanych języku Python. Przykładowe umieściliśmy poniżej. Sprawdzaliśmy w nich wszystkie konieczne warunki, które powinny spełniać poprawne dane, a następnie przy wykorzystaniu funkcji pseudolosowych wygenerowaliśmy dane dla każdej z tabel. Na końcu za pomocą zapytań SQL sprawdziliśmy ich poprawność w bazie danych.

Sprawdzanie poprawności webinariów:

```
import csv, os
from datetime import datetime, timedelta

os.chdir(os.path.dirname(__file__))

webinars_csv = '../Tables Data/Webinars.csv'
user_roles_csv = '../Tables Data/UserRoles.csv'
employees_csv = '../Tables Data/Employees.csv'
Translated_languages_csv = '../Tables Data/TranslatedLanguage.csv'


def wczytaj_liste_z_csv(plik):
    with open(plik, mode = 'r', encoding = 'utf-8') as file:
```

```
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]


webinars = wczytaj_liste_z_csv(webinars_csv)
user_roles = wczytaj_liste_z_csv(user_roles_csv)
employees = wczytaj_liste_z_csv(employees_csv)
translated_languages = wczytaj_liste_z_csv(Translated_languages_csv)
translations = dict()
teaching = dict()
flags = [True] * 6


def check_if_translator_employed_before_start_date():
    for webinar in webinars:
        start_date = datetime.strptime(webinar[7], '%d.%m.%Y %H:%M')
        translator_id = webinar[6]
        translations[translator_id] = None
        for employee in employees:
            if employee[0] == translator_id:
                employment_date = datetime.strptime(employee[1], '%Y-%m-
%d')
                if employment_date > start_date:
                    print(f"Translator {translator_id} was employed
after start date of webinar {webinar[1]}")
                    flags[0] = False
                break


def check_if_teacher_employed_before_start_date():
    for webinar in webinars:
        start_date = datetime.strptime(webinar[7], '%d.%m.%Y %H:%M')
        teacher_id = webinar[3]
        teaching[teacher_id] = None
        for employee in employees:
            if employee[0] == teacher_id:
                employment_date = datetime.strptime(employee[1], '%Y-%m-
%d')
                if employment_date > start_date:
                    print(f"Teacher {teacher_id} was employed after
start date of webinary {webinar[1]}")
                    flags[1] = False
                break
```

```
def
check_if_there_arent_two_translations_at_the_same_time_for_one_employee(
):
    for webinar in webinars:
        start_date = datetime.strptime(webinar[7], '%d.%m.%Y %H:%M')
        end_date = datetime.strptime(webinar[8], '%d.%m.%Y %H:%M')
        translator_id = webinar[6]
        teacher_id = webinar[3]
        if translator_id != "" and (translations[translator_id] == None
or translations[translator_id] <= start_date):
            translations[translator_id] = end_date
        elif translator_id != "":
            print(f"Translator {translator_id} has two translations at
the same time for webinary {webinar[1]}")
            flags[2] = False
        if teaching[teacher_id] == None or teaching[teacher_id] <=
start_date:
            teaching[teacher_id] = end_date
        else:
            print(f"Teacher {teacher_id} has two translations at the
same time for webinary {webinar[1]}")
            flags[3] = False


def check_if_start_date_is_before_end_date():
    for webinar in webinars:
        start_date = datetime.strptime(webinar[7], '%d.%m.%Y %H:%M')
        end_date = datetime.strptime(webinar[8], '%d.%m.%Y %H:%M')
        if start_date > end_date:
            print(f"Start date is after end date for webinary
{webinar[1]}")
            flags[4] = False


def check_if_duration_is_between_45_and_180():
    for webinar in webinars:
        start_date = datetime.strptime(webinar[7], '%d.%m.%Y %H:%M')
        end_date = datetime.strptime(webinar[8], '%d.%m.%Y %H:%M')
        duration = end_date - start_date
        if duration < timedelta(minutes = 45) or duration >
timedelta(minutes = 180):
            print(
                f"Duration of webinar {webinar[1]} is not between 45 and
180 minutes, webinar no {webinar[0]} - {duration}")
            flags[5] = False
```

```
if __name__ == "__main__":
    check_if_translator_employed_before_start_date()
    check_if_teacher_employed_before_start_date()

check_if_there_arent_two_translations_at_the_same_time_for_one_employee(
)
    check_if_start_date_is_before_end_date()
    check_if_duration_is_between_45_and_180()
    if all(flags):
        print("ALl data correct")
    else:
        print("Something incorrect")
```

Sprawdzanie poprawności spotkań w ramach kursów:

```
import os, csv, datetime

os.chdir(os.path.dirname(__file__))

def wczytaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]

meetings_csv = "../Tables Data/a_new_meetings.csv"
courses_csv = "../Tables Data/courses_without_students_limit.csv"
modules_csv = "../Tables Data/new_modules.csv"
employees_csv = "../Tables Data/Employees.csv"
webinars_csv = "../Tables Data/Webinars.csv"

meetings = wczytaj_liste_z_csv(meetings_csv)
courses = wczytaj_liste_z_csv(courses_csv)
modules = wczytaj_liste_z_csv(modules_csv)
employees = wczytaj_liste_z_csv(employees_csv)
webinars = wczytaj_liste_z_csv(webinars_csv)

users_meetings = {}
for module in modules:
    if module[4] not in users_meetings:
        users_meetings[module[4]] = []
    if module[6] != "" and module[6] not in users_meetings:
        users_meetings[module[6]] = []
```

```
modules_meetings = {}
for meeting in meetings:
    if meeting[1] not in modules_meetings.keys():
        modules_meetings[meeting[1]] = []
    modules_meetings[meeting[1]].append((meeting[2], meeting[3]))

def add_meetings_date_to_users():
    for module in modules_meetings.keys():
        for meeting in modules_meetings[module]:
            users_meetings[modules[int(module)-
1][4]].append((meeting[0], meeting[1]))
            if modules[int(module)-1][6] != "":
                users_meetings[modules[int(module)-
1][6]].append((meeting[0], meeting[1]))

add_meetings_date_to_users()

def make_dict_with_employees():
    employees_dict = {}
    for employee in employees:
        employees_dict[employee[0]] = employee[1]
    return employees_dict
employees_dict = make_dict_with_employees()

def check_if_date_is_ok(user_id, start_date, end_date):
    cnt = 0
    for meeting in users_meetings[user_id]:
        meeting_start = meeting[0]
        meeting_end = meeting[1]
        if not (end_date <= meeting_start or start_date >=
meeting_end):
            cnt += 1
        if cnt>1:
            return False
    return True

def check_if_employee_has_more_than_one_meeting_at_the_same_time():
    for user in users_meetings.keys():
        for meeting in users_meetings[user]:
            if not check_if_date_is_ok(user, meeting[0], meeting[1]):
                print(user, meeting[0], meeting[1])
                print("User has more than one meeting at the same
time")
                return False
```

```
    return True


def check_if_all_meetings_are_between_monday_and_thursday():
    for module in modules_meetings.keys():
        for meeting in modules_meetings[module]:
            start_date = datetime.datetime.strptime(meeting[0], '%Y-%m-
%d %H:%M')
            if start_date.weekday() > 3:
                print(module, meeting[0])
                print("Meeting is not between Monday and Thursday")
                return False
    return True

def check_if_all_meetings_are_between_2021_and_2026():
    for module in modules_meetings.keys():
        for meeting in modules_meetings[module]:
            start_date = datetime.datetime.strptime(meeting[0], '%Y-%m-
%d %H:%M')
            if start_date.year < 2021 or start_date.year > 2026:
                print(module, meeting[0])
                print("Meeting is not between 2021 and 2026")
                return False
    return True


def check_if_employee_employed_before_course_start_date():
    for module in modules:
        lecturer_id = module[4]
        translator_id = module[6]
        lecturer_start_date =
datetime.datetime.strptime(employees_dict[lecturer_id], '%Y-%m-%d')
        if translator_id != "":
            translator_start_date =
datetime.datetime.strptime(employees_dict[translator_id], '%Y-%m-%d')
        else:
            translator_start_date = None
        for meeting in modules_meetings[module[0]]:
            meeting_start_date = datetime.datetime.strptime(meeting[0],
'%Y-%m-%d %H:%M')
            if lecturer_start_date > meeting_start_date:
                print(module)
                print("Lecturer was employed after meeting start date")
                return False
            if translator_start_date is not None and
```

```
translator_start_date and translator_start_date > meeting_start_date:
                print(module)
                print("Translator was employed after meeting start
date")
                return False
    return True

def check_if_employee_dont_have_webinar_at_the_same_time():
    for webinar in webinars:
        webinar_start_date = webinar[7]
        webinar_end_date = webinar[8]
        if webinar[3] in users_meetings.keys():
            if not check_if_date_is_ok(webinar[3], webinar_start_date,
webinar_end_date):
                print(webinar)
                print("User has more than one meeting at the same
time")
                break
        if webinar[6]!= "" and webinar[6] in users_meetings.keys():
            if not check_if_date_is_ok(webinar[6], webinar_start_date,
webinar_end_date):
                print(webinar)
                print("User has more than one meeting at the same
time")
                return False
        return True

def make_dict_with_beginning_and_end_of_course():
    dict = {}
    for course in courses:
        dict[course[0]] = None
    return dict
courses_dict = make_dict_with_beginning_and_end_of_course()

def check_if_course_is_no_more_than_14_days():
    for module in modules:
        course_id = module[1]
        for meeting in modules_meetings[module[0]]:
            meeting_date = datetime.datetime.strptime(meeting[0], '%Y-
%m-%d %H:%M')
            if courses_dict[course_id] is None:
                courses_dict[course_id] = (meeting_date, meeting_date)
            else:
                if meeting_date < courses_dict[course_id][0]:
                    courses_dict[course_id] = (meeting_date,
```

```
courses_dict[course_id][1])
                if meeting_date > courses_dict[course_id][1]:
                    courses_dict[course_id] =
(courses_dict[course_id][0], meeting_date)
    for course in courses_dict.keys():
        if (courses_dict[course][1] - courses_dict[course][0]).days >
14:
            print(course)
            print("Course is longer than 14 days")
            return False
    return True

def check_if_all_modules_have_at_least_one_meeting():
    for module in modules:
        if module[0] not in modules_meetings.keys():
            print(module)
            print("Module has no meetings")
            return False
    return True

def check_if_modules_type_2_has_at_leat_2_meetings():
    for module in modules:
        if module[3] == "2":
            if module[0] not in modules_meetings.keys() or
len(modules_meetings[module[0]]) < 2:
                print(module)
                print("Module type 2 has less than 2 meetings")
                return False
    return True

def check_if_all_meetings_are_correct():
    if (check_if_modules_type_2_has_at_leat_2_meetings and
check_if_all_meetings_are_between_monday_and_thursday() and
check_if_all_meetings_are_between_2021_and_2026() and
check_if_employee_dont_have_webinar_at_the_same_time() and
check_if_employee_has_more_than_one_meeting_at_the_same_time and
check_if_course_is_no_more_than_14_days() and
check_if_all_modules_have_at_least_one_meeting() and
check_if_modules_type_2_has_at_leat_2_meetings()):
        print("All meetings are correct")
    else:
        print("Meetings are not correct")

check_if_all_meetings_are_correct()
```

Generator zamówień kursów:

```
import csv, random, os, string
from datetime import datetime, timedelta

os.chdir(os.path.dirname(__file__))

def wczystaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        next(reader)
        return [row for row in reader]

def wczytaj_liste_z_csv_z_naglowkiem(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        return [row for row in reader]


course_modules_csv = "../Tables Data/CourseModules.csv"
course_modules = wczystaj_liste_z_csv(course_modules_csv)
courses_csv = "../Tables Data/Courses.csv"
courses = wczystaj_liste_z_csv(courses_csv)
online_course_meetings_csv = "../Tables Data/OnlineCourseMeeting.csv"
online_course_meetings =
wczystaj_liste_z_csv(online_course_meetings_csv)
stationary_course_meetings_csv = "../Tables
Data/StationaryCourseMeeting.csv"
stationary_course_meetings =
wczystaj_liste_z_csv(stationary_course_meetings_csv)
order_details_csv = "../Tables Data/OrderDetails.csv"
order_details = wczystaj_liste_z_csv(order_details_csv)
users_csv = "../Tables Data/Users.csv"
users = wczystaj_liste_z_csv(users_csv)
orders_csv = "../Tables Data/Orders.csv"
orders = wczystaj_liste_z_csv(orders_csv)
studies_csv = "../Tables Data/Studies.csv"
studies = wczystaj_liste_z_csv(studies_csv)
subjects_csv = "../Tables Data/Subjects.csv"
subjects = wczystaj_liste_z_csv(subjects_csv)
webinars_csv = "../Tables Data/Webinars.csv"
webinars = wczystaj_liste_z_csv(webinars_csv)
```

```
study_meetings_csv = "../Tables Data/StudyMeetings.csv"
study_meetings = wczystaj_liste_z_csv(study_meetings_csv)
starts_and_ends_csv = "./Result_68.csv"
dates = wczytaj_liste_z_csv_z_naglowkiem(starts_and_ends_csv)



def check_if_date_is_ok(user_id, start_date, end_date):
    cnt = 0
    for meeting in users_meetings[user_id]:
        meeting_start = datetime.strptime(meeting[0][:16],"%Y-%m-%d
%H:%M")
        meeting_end = datetime.strptime(meeting[1][:16],"%Y-%m-%d
%H:%M")
        if not (end_date <= meeting_start or start_date >=
meeting_end):
            cnt += 1
        if cnt>1:
            return False
    return True


users_meetings = {}

#dodaj wszystkie study_meetings i kursy z orderów do usera
for user in users:
    users_meetings[user[0]] = []
    for order in orders:
        if order[1] == user[0]:
            for order_detail in order_details:
                if order_detail[1]==order[0]:
                    if(order_detail[3] == "3"):#studies
                        for subject in subjects:
                            if subject[1]==order_detail[2]:
                                for study_meeting in study_meetings:
                                    if study_meeting[1]==subject[0]:

users_meetings[user[0]].append((study_meeting[6], study_meeting[7]))
                    elif(order_detail[3] == "2"):#courses
                        for course_module in course_modules:
                            if course_module[1]==order_detail[2]:
                                for stationary_course_meeting in
stationary_course_meetings:
                                    if
stationary_course_meeting[1]==course_modules[0]:
```

```
users_meetings[user[0]].append((stationary_course_meeting[3],
stationary_course_meeting[4]))
                                for online_course_meeting in
online_course_meetings:
                                    if
online_course_meeting[1]==course_modules[0]:

users_meetings[user[0]].append((online_course_meeting[2],
online_course_meeting[3]))
                    elif(order_detail[3] == "4"):#pojedyncze
spotkania studyjne
                        meeting =
study_meetings[int(order_detail[2])-1]

users_meetings[user[0]].append((study_meeting[6], study_meeting[7]))

# print(users_meetings)


#dodaj wszytskie prowadzone spotkania do userów
for user in users:
    for module in course_modules:
        if module[3]!='2' and (module[4]==user[0] or
module[5]==user[0]):
            for online_course_meeting in online_course_meetings:
                if online_course_meeting[1]==module[0]:

users_meetings[user[0]].append((online_course_meeting[2],
online_course_meeting[3]))
            for stationary_course_meeting in
stationary_course_meetings:
                if stationary_course_meeting[1]==module[0]:

users_meetings[user[0]].append((stationary_course_meeting[3],
stationary_course_meeting[4]))
    for webinar in webinars:
        if webinar[3]==user[0] or webinar[6]==user[0]:
            users_meetings[user[0]].append((webinar[7], webinar[8]))
    for meeting in study_meetings:
        if meeting[2]==user[0] or meeting[9]==user[0]:
            users_meetings[user[0]].append((meeting[6], meeting[7]))
```

```
#pogrupuj userów po mieście, pogrupować kursy
users_by_city = {}
courses_by_city = {}
courses_cities = {}
for user in users:
    if user[4] not in users_by_city:
        users_by_city[user[4]] = []
    users_by_city[user[4]].append(user[0])
for course in courses:
    for module in course_modules:
        if module[1]==course[0] and module[3]!="2" and
module[3]!="3":
            if users[int(module[4])-1][4] not in courses_by_city:
                courses_by_city[users[int(module[4])-1][4]] = []
            courses_by_city[users[int(module[4])-
1][4]].append(course[0])
            courses_cities[course[0]] = users[int(module[4])-1][4]


# # print(stationary_course_meetings)
def generate_course_and_webinars_orders():
    global webinars, courses, course_modules,
stationary_course_meetings, online_course_meetings, users_by_city,
courses_cities, course_dates
    course_dates = {}
    for i, course in enumerate(courses):
        course_dates[str(i+1)] =
(datetime.strptime(dates[i][1][:16],"%Y-%m-%d %H:%M"),
datetime.strptime(dates[i][2][:16],"%Y-%m-%d %H:%M"))

    # print(course_dates)
    # for course in courses:
    #     start = None
    #     end = None
    #     cnt=0
    #     last1 = 0
    #     last2 = 0
    #     for module in course_modules:
    #         for i in range(last1, len(stationary_course_meetings)):
    #             stationary_course_meeting =
stationary_course_meetings[i]
    #             print(stationary_course_meeting)
    #             # print(cnt)
    #             # cnt+=1
```

```
    #             if stationary_course_meeting[1]==module[0]:
    #                 if start==None or
start>datetime.strptime(stationary_course_meeting[3][:16],"%Y-%m-%d
%H:%M"):
    #                     start =
datetime.strptime(stationary_course_meeting[3][:16], "%Y-%m-%d
%H:%M")
    #                 if end==None or
end<datetime.strptime(stationary_course_meeting[4][:16],"%Y-%m-%d
%H:%M"):
    #                     end =
datetime.strptime(stationary_course_meeting[4][:16], "%Y-%m-%d
%H:%M")
    #             if stationary_course_meeting[1]>module[0]:
    #                 last1 = i
    #                 break
    #         for i in range(last2,len(online_course_meetings)):
    #             online_course_meeting = online_course_meetings[i]
    #             if online_course_meeting[1]==module[0]:
    #                 if start==None or
start>datetime.strptime(online_course_meeting[2][:16],"%Y-%m-%d
%H:%M"):
    #                     start =
datetime.strptime(online_course_meeting[2][:16], "%Y-%m-%d %H:%M")
    #                 if end==None or
end<datetime.strptime(online_course_meeting[3][:16],"%Y-%m-%d
%H:%M"):
    #                     end =
datetime.strptime(online_course_meeting[3][:16],"%Y-%m-%d %H:%M")
    #             if online_course_meeting[1]>module[0]:
    #                 last2 = i
    #                 break
    #     course_dates[course[0]] = (start, end)

#     # print(course_dates)


    webinars_orders = []
    course_orders = []
    all_users = [user for sublist in users_by_city.values() for user
in sublist]
    # print(all_users)
    #tworz ordersy sprawdzajac czy dodanie koeljnej aktywności z
niczym innym nie koliduje
    # print(courses)
```

```
    for course in courses:
        if course[0] in courses_cities.keys():
            city = courses_cities[course[0]]
            limit = random.randint(10, int(course[5]))
            counter = 0
            random.shuffle(users_by_city[city])
            for user in (users_by_city[city]):

                start_date = course_dates[course[0]][0]
                print(start_date)
                end_date = course_dates[course[0]][1]
                if check_if_date_is_ok(user, start_date, end_date):
                    string_date = start_date.strftime("%Y-%m-%d
%H:%M")

course_orders.append([course[0],'2',course[4],user,string_date,'stati
onary/hybrid'])
                    counter+=1
                if counter==limit:
                    break
        else:
            limit = random.randint(10, 20)
            counter = 0
            random.shuffle(all_users)
            for user in (all_users):
                start_date = course_dates[course[0]][0]
                end_date = course_dates[course[0]][1]
                if check_if_date_is_ok(user, start_date, end_date):
                    string_date = start_date.strftime("%Y-%m-%d
%H:%M")

course_orders.append([course[0],'2',course[4],user,string_date,'onlin
e'])
                    counter+=1
                if counter==limit:
                    break


    for webinar in webinars:
        limit = random.randint(10, 20)
        counter = 0
        random.shuffle(all_users)
        for user in (all_users):
            start_date = datetime.strptime(webinar[7][:16],"%Y-%m-%d
%H:%M")
```

```
            end_date = datetime.strptime(webinar[8][:16],"%Y-%m-%d
%H:%M")
            if check_if_date_is_ok(user, start_date, end_date):

webinars_orders.append([webinar[0],'1',webinar[4],user,start_date,'we
binar'])
                counter+=1
            if counter==limit:
                break

    output_file = "course_and_webinars_orders.csv"
    # with open(output_file, mode='w', encoding='utf-8', newline='')
as file:
    #     writer = csv.writer(file)
    #     writer.writerow(["acivity_id", "acivity_type", "price",
"user_id", "start_date", "type"])
    #     for course_order in course_orders:
    #         writer.writerow(course_order)
    #     for webinar_order in webinars_orders:
    #         writer.writerow(webinar_order)

generate_course_and_webinars_orders()
# ##############################################

course_and_webinars_orders_detais_csv =
"./course_and_webinars_orders.csv"
course_and_webinars_orders =
wczystaj_liste_z_csv(course_and_webinars_orders_detais_csv)
generate_course_and_webinars_orders()

def generate_date(start_date):
    delta_days = random.randint(10, 40)
    generated_date = start_date - timedelta(days=delta_days)
    return generated_date.strftime("%Y-%m-%d %H:%M")

def generate_pamyment_link():
    return
f"https://www.paypal.com/pl/home/{''.join(random.choices(string.ascii
_letters, k=10))}"

course_and_webinars_orders.sort(key=lambda x:
datetime.strptime(x[4][:16], "%Y-%m-%d %H:%M"))
course_and_webinars_orders.sort(key=lambda x: x[3])

i = 0
```

```
curr_order_id = 584
curr_detail_id = 584
orders_list = []
advances_to_add = []
order_details_to_add = []
while i<len(course_and_webinars_orders):
    group = []
    j = i

    while j<len(course_and_webinars_orders) and
course_and_webinars_orders[i][3]==course_and_webinars_orders[j][3]
and (datetime.strptime(course_and_webinars_orders[j][4][:16], "%Y-%m-
%d %H:%M") - datetime.strptime(course_and_webinars_orders[i][4][:16],
"%Y-%m-%d %H:%M")).days < 60:
        group.append(course_and_webinars_orders[j])
        j+=1
    order_id = curr_order_id
    curr_order_id+=1
    user_id = course_and_webinars_orders[i][3]
    order_date =
generate_date(datetime.strptime(course_and_webinars_orders[i][4][:16]
, "%Y-%m-%d %H:%M"))
    if order_date>datetime.now().strftime("%Y-%m-%d %H:%M"):
        while j<len(course_and_webinars_orders) and
course_and_webinars_orders[i][3]==course_and_webinars_orders[j][3]:
            j+=1
        i = j
        continue

    orders_list.append([order_id, user_id, order_date, 0, None,
generate_pamyment_link()])
    for k in range(i,j):
        detail = course_and_webinars_orders[k]
        date_of_order = datetime.strptime(order_date, "%Y-%m-%d
%H:%M")
        price = detail[2]
        if price != None and
date_of_order<datetime.now()+timedelta(days=11):
            payment_date = date_of_order +
timedelta(minutes=random.randint(1, 10))
            status = 'udane'
        else:
            payment_date = None
            status = None
        order_details_to_add.append([curr_detail_id, order_id,
```

```
detail[0], detail[1], price, payment_date, status])
        if detail[1]=='2':
            payment_date = date_of_order +
timedelta(minutes=random.randint(1, 10))
            status = 'udane'
            advances_to_add.append([curr_detail_id,
round(0.1*float(price),2), payment_date, status])
        curr_detail_id+=1
    i = j

output_file = "courses_and_webinars_orders_after_groupping_1.csv"
with open(output_file, mode='w', encoding='utf-8', newline='') as
file:
    writer = csv.writer(file)
    writer.writerow(["orderId", "userId", "orderDate",
"paymentDeffered", "deffereDate", "paymentLink"])
    for order in orders_list:
        writer.writerow(order)


output_file = "advances_to_add.csv"
with open(output_file, mode='w', encoding='utf-8', newline='') as
file:
    writer = csv.writer(file)
    writer.writerow(["detailId", "price", "paymentDate", "status"])
    for advance in advances_to_add:
        writer.writerow(advance)

output_file_1 = "order_details_to_add.csv"
with open(output_file_1, mode='w', encoding='utf-8', newline='') as
file:
    writer = csv.writer(file)
    writer.writerow(["orderId", "activityId", "activityType",
"price", "paymentDate", "status"])
    for detail in order_details_to_add:
        writer.writerow(detail)
```

Generator modułów kursów:

```
import os, csv, datetime
```

```
os.chdir(os.path.dirname(__file__))

def wczytaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]

meetings_csv = "../Tables Data/a_new_meetings.csv"
courses_csv = "../Tables Data/courses_without_students_limit.csv"
modules_csv = "../Tables Data/new_modules.csv"
employees_csv = "../Tables Data/Employees.csv"
webinars_csv = "../Tables Data/Webinars.csv"

meetings = wczytaj_liste_z_csv(meetings_csv)
courses = wczytaj_liste_z_csv(courses_csv)
modules = wczytaj_liste_z_csv(modules_csv)
employees = wczytaj_liste_z_csv(employees_csv)
webinars = wczytaj_liste_z_csv(webinars_csv)

users_meetings = {}
for module in modules:
    if module[4] not in users_meetings:
        users_meetings[module[4]] = []
    if module[6] != "" and module[6] not in users_meetings:
        users_meetings[module[6]] = []

modules_meetings = {}
for meeting in meetings:
    if meeting[1] not in modules_meetings.keys():
        modules_meetings[meeting[1]] = []
    modules_meetings[meeting[1]].append((meeting[2], meeting[3]))

def add_meetings_date_to_users():
    for module in modules_meetings.keys():
        for meeting in modules_meetings[module]:
            users_meetings[modules[int(module)-
1][4]].append((meeting[0], meeting[1]))
            if modules[int(module)-1][6] != "":
                users_meetings[modules[int(module)-
1][6]].append((meeting[0], meeting[1]))

add_meetings_date_to_users()

def make_dict_with_employees():
```

```
    employees_dict = {}
    for employee in employees:
        employees_dict[employee[0]] = employee[1]
    return employees_dict
employees_dict = make_dict_with_employees()

def check_if_date_is_ok(user_id, start_date, end_date):
    cnt = 0
    for meeting in users_meetings[user_id]:
        meeting_start = meeting[0]
        meeting_end = meeting[1]
        if not (end_date <= meeting_start or start_date >=
meeting_end):
            cnt += 1
        if cnt>1:
            return False
    return True

def check_if_employee_has_more_than_one_meeting_at_the_same_time():
    for user in users_meetings.keys():
        for meeting in users_meetings[user]:
            if not check_if_date_is_ok(user, meeting[0], meeting[1]):
                print(user, meeting[0], meeting[1])
                print("User has more than one meeting at the same
time")
                return False
    return True


def check_if_all_meetings_are_between_monday_and_thursday():
    for module in modules_meetings.keys():
        for meeting in modules_meetings[module]:
            start_date = datetime.datetime.strptime(meeting[0], '%Y-%m-
%d %H:%M')
            if start_date.weekday() > 3:
                print(module, meeting[0])
                print("Meeting is not between Monday and Thursday")
                return False
    return True

def check_if_all_meetings_are_between_2021_and_2026():
    for module in modules_meetings.keys():
        for meeting in modules_meetings[module]:
            start_date = datetime.datetime.strptime(meeting[0], '%Y-%m-
%d %H:%M')
```

```
            if start_date.year < 2021 or start_date.year > 2026:
                print(module, meeting[0])
                print("Meeting is not between 2021 and 2026")
                return False
    return True


def check_if_employee_employed_before_course_start_date():
    for module in modules:
        lecturer_id = module[4]
        translator_id = module[6]
        lecturer_start_date =
datetime.datetime.strptime(employees_dict[lecturer_id], '%Y-%m-%d')
        if translator_id != "":
            translator_start_date =
datetime.datetime.strptime(employees_dict[translator_id], '%Y-%m-%d')
        else:
            translator_start_date = None
        for meeting in modules_meetings[module[0]]:
            meeting_start_date = datetime.datetime.strptime(meeting[0],
'%Y-%m-%d %H:%M')
            if lecturer_start_date > meeting_start_date:
                print(module)
                print("Lecturer was employed after meeting start date")
                return False
            if translator_start_date is not None and
translator_start_date and translator_start_date > meeting_start_date:
                print(module)
                print("Translator was employed after meeting start
date")
                return False
    return True

def check_if_employee_dont_have_webinar_at_the_same_time():
    for webinar in webinars:
        webinar_start_date = webinar[7]
        webinar_end_date = webinar[8]
        if webinar[3] in users_meetings.keys():
            if not check_if_date_is_ok(webinar[3], webinar_start_date,
webinar_end_date):
                print(webinar)
                print("User has more than one meeting at the same
time")
                break
        if webinar[6]!= "" and webinar[6] in users_meetings.keys():
```

```
            if not check_if_date_is_ok(webinar[6], webinar_start_date,
webinar_end_date):
                print(webinar)
                print("User has more than one meeting at the same
time")
                return False
        return True

def make_dict_with_beginning_and_end_of_course():
    dict = {}
    for course in courses:
        dict[course[0]] = None
    return dict
courses_dict = make_dict_with_beginning_and_end_of_course()

def check_if_course_is_no_more_than_14_days():
    for module in modules:
        course_id = module[1]
        for meeting in modules_meetings[module[0]]:
            meeting_date = datetime.datetime.strptime(meeting[0], '%Y-
%m-%d %H:%M')
            if courses_dict[course_id] is None:
                courses_dict[course_id] = (meeting_date, meeting_date)
            else:
                if meeting_date < courses_dict[course_id][0]:
                    courses_dict[course_id] = (meeting_date,
courses_dict[course_id][1])
                if meeting_date > courses_dict[course_id][1]:
                    courses_dict[course_id] =
(courses_dict[course_id][0], meeting_date)
    for course in courses_dict.keys():
        if (courses_dict[course][1] - courses_dict[course][0]).days >
14:
            print(course)
            print("Course is longer than 14 days")
            return False
    return True

def check_if_all_modules_have_at_least_one_meeting():
    for module in modules:
        if module[0] not in modules_meetings.keys():
            print(module)
            print("Module has no meetings")
            return False
    return True
```

```
def check_if_modules_type_2_has_at_leat_2_meetings():
    for module in modules:
        if module[3] == "2":
            if module[0] not in modules_meetings.keys() or
len(modules_meetings[module[0]]) < 2:
                print(module)
                print("Module type 2 has less than 2 meetings")
                return False
    return True

def check_if_all_meetings_are_correct():
    if (check_if_modules_type_2_has_at_leat_2_meetings and
check_if_all_meetings_are_between_monday_and_thursday() and
check_if_all_meetings_are_between_2021_and_2026() and
check_if_employee_dont_have_webinar_at_the_same_time() and
check_if_employee_has_more_than_one_meeting_at_the_same_time and
check_if_course_is_no_more_than_14_days() and
check_if_all_modules_have_at_least_one_meeting() and
check_if_modules_type_2_has_at_leat_2_meetings()):
        print("All meetings are correct")
    else:
        print("Meetings are not correct")

check_if_all_meetings_are_correct()
```

Generowanie spotkań w ramach kursu:

```
import random, csv, os

os.chdir(os.path.dirname(__file__))

user_roles_csv = "UserRoles.csv"
courses_csv = "courses_without_students_limit.csv"
translated_languages_csv = "TranslatedLanguage.csv"
users_csv = "Users.csv"


def wczytaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
```

```
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]

user_roles = wczytaj_liste_z_csv(user_roles_csv)
courses = wczytaj_liste_z_csv(courses_csv)
translated_languages = wczytaj_liste_z_csv(translated_languages_csv)
users = wczytaj_liste_z_csv(users_csv)
output_csv = 'courses_modules.csv'

lecturers = []
translators = []
for user_role in user_roles:
    if user_role[1] == "7":
        lecturers.append(user_role[0])
    if user_role == "2":
        translators.append(user_role[0])

users_cities = {}
cities_users = {}
cities_translators = {}
cities_teachers = {}
for user in users:
    city = user[4]
    users_cities[user[0]] = city
    if city not in cities_users.keys():
        cities_users[city] = []
    cities_users[city].append(user[0])
    if user[0] in lecturers:
        if city not in cities_teachers.keys():
            cities_teachers[city] = []
        cities_teachers[city].append(user[0])
    if user[0] in translators:
        if city not in cities_translators.keys():
            cities_teachers[city] = []
        cities_translators[city].append(user[0])

dict_of_traslators_for_certain_language_in_certain_city = {}
translators_for_languages = {}
for language in translated_languages:
    if language[1] not in translators_for_languages.keys():
        translators_for_languages[language[1]] = []
    city = users_cities[language[0]]
    if city not in
dict_of_traslators_for_certain_language_in_certain_city.keys():
```

```
        dict_of_traslators_for_certain_language_in_certain_city[city]
= {}
    if language[1] not in
dict_of_traslators_for_certain_language_in_certain_city[city].keys():

dict_of_traslators_for_certain_language_in_certain_city[city][languag
e[1]] = []

dict_of_traslators_for_certain_language_in_certain_city[city][languag
e[1]].append(language[0])
    translators_for_languages[language[1]].append(language[0])
all_possibilities_of_translations = len(translated_languages)

def generate_language_and_translator():
    random_value = random.random()
    if random_value<0.7:
        language = 1 #70% chances for polish
        return (1, None)
    top_limit = 0.7
    for language in translators_for_languages.keys():
        top_limit += len(translators_for_languages[language]) /
all_possibilities_of_translations * (1-0.7)
        if random_value <= top_limit:
            translator =
random.choice(translators_for_languages[language])
            return (language, translator)

def generate_language_and_translator_from_city(city):
    while True:
        random_value = random.random()
        if random_value<0.7:
            language = 1 #70% chances for polish
            return (1, None)
        top_limit = 0.7
        for language in translators_for_languages.keys():
            top_limit += len(translators_for_languages[language]) /
all_possibilities_of_translations * (1-0.7)
            if random_value <= top_limit:
                if language in
dict_of_traslators_for_certain_language_in_certain_city[city].keys():
                    translator =
random.choice(dict_of_traslators_for_certain_language_in_certain_city
[city][language])
                    return (language, translator)
```

```
def generate_teacher_from_city(city):
    if city is None:
        return random.choice(lecturers)
    else:
        return random.choice(cities_teachers[city])



def generate_course_modules():
    module_number = 1
    course_modules = []
    for course in courses:
        city = None
        course_id = course[0]
        number_of_modules = random.randint(3, 8)
        for i in range(number_of_modules):
            module_id = module_number
            module_number += 1
            module_name = f"Moduł {i + 1} kursu '{course[1]}'"
            module_description = f"Część {i+1} kursu '{course[1]}'"
            module_type = random.randint(1, 4)
            if module_type == 2 or module_type == 3:\
                #for online courses it doen't matters from which city
the employee is
                module_language, module_translator =
generate_language_and_translator()
                teacher_id = random.choice(lecturers)
            else:
                if city is None:
                    module_language, module_translator =
generate_language_and_translator()
                    if module_language != 1:
                        city = users_cities[module_translator]
                else:
                    module_language, module_translator =
generate_language_and_translator_from_city(city)
                teacher_id = generate_teacher_from_city(city)
            course_modules.append([module_id, course_id, module_name,
module_type, teacher_id, module_language, module_translator])
    return course_modules



course_modules = generate_course_modules()
with open(output_csv, mode='w', newline='', encoding='utf-8') as
```

```
 file:
     writer = csv.writer(file)
     writer.writerow(["ModuleID", "CourseID", "ModuleName",
"ModuleType", "LecturerID", "LanguageID", "TranslatorID"])
     writer.writerows(course_modules)
```

Dodawanie sal do kursów:

```
import random, csv, os

os.chdir(os.path.dirname(__file__))

meetings_csv = "../Tables Data/StationaryCourseMeeting.csv"
courses_csv = "../Tables Data/Courses.csv"
modules_csv = "../Tables Data/CourseModules.csv"
users = "../Tables Data/Users.csv"
rooms = "../Tables Data/Rooms.csv"


def wczytaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]

meetings = wczytaj_liste_z_csv(meetings_csv)
courses = wczytaj_liste_z_csv(courses_csv)
modules = wczytaj_liste_z_csv(modules_csv)
users = wczytaj_liste_z_csv(users)
rooms = wczytaj_liste_z_csv(rooms)


def make_dict_with_cities_rooms():
    cities_rooms = {}
    for room in rooms:
        city = room[3]
        if city not in cities_rooms.keys():
            cities_rooms[city] = []
        cities_rooms[city].append(room[0])
    return cities_rooms
```

```
cities_rooms = make_dict_with_cities_rooms()

cnt = 0
for meeting in meetings:
    module = modules[int(meeting[1])-1]
    course_id = module[1]
    lecturer = users[int(module[4])-1]
    city = lecturer[4]
    if city in cities_rooms.keys():
        room = random.choice(cities_rooms[city])
    else:
        cnt += 1
        print(course_id)
    meeting.insert(2, room)

output = 'new_stationary_meetings.csv'
with open(output, mode='w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerow(["MeetingID", "ModuleID", "RoomID", "StartDate",
"EndDate"])
    for meeting in meetings:
        writer.writerow(meeting)
```

Generowanie cen kursów:

```
import random, os, csv
from datetime import datetime, timedelta

os.chdir(os.path.dirname(__file__))

courses_csv = "../Tables Data/courses_without_students_limit.csv"
modules_csv = "../Tables Data/CourseModules.csv"
meetings_csv = "../Tables Data/a_new_meetings.csv"
output_csv = "../Tables Data/courses_with_price.csv"

def wczytaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]

courses = wczytaj_liste_z_csv(courses_csv)
```

```
modules = wczytaj_liste_z_csv(modules_csv)
meetings = wczytaj_liste_z_csv(meetings_csv)

output_courses = []

modules_meetings = {}
for meeting in meetings:
    if meeting[1] not in modules_meetings.keys():
        modules_meetings[meeting[1]] = []
    modules_meetings[meeting[1]].append(meeting)

courses_modules = {}
for module in modules:
    if module[1] not in courses_modules.keys():
        courses_modules[module[1]] = []
    courses_modules[module[1]].append(module[0])

for course in courses:
    time_of_course_meeetings = timedelta(minutes = 0)
    for module in courses_modules[course[0]]:
        for meeting in modules_meetings[module]:
            time_of_course_meeetings += datetime.strptime(meeting[3],
"%Y-%m-%d %H:%M") - datetime.strptime(meeting[2], "%Y-%m-%d %H:%M")
    price = round(time_of_course_meeetings.total_seconds() / 3600 * 50)
    print(price, time_of_course_meeetings.total_seconds() / 3600)
    output_courses.append([course[0], course[1], course[2], course[3],
course[4], price])

with open(output_csv, mode='w', encoding='utf-8', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["CourseID", "CourseName", "CourseCoordinatorID",
"CourseDescription", "CoursePrice"])
    for row in output_courses:
        writer.writerow(row)
```

Generowanie limitu kursów

```
import csv, random, os

os.chdir(os.path.dirname(__file__))

rooms_csv = "../Tables Data/Rooms.csv"
courses_csv = "../Tables Data/courses_with_price.csv"
```

```
output_csv = "../Tables Data/Courses1.csv"
course_modules_csv = "../Tables Data/CourseModules.csv"
course_stationary_meetings_csv = "../Tables
Data/StationaryCourseMeeting.csv"

def wczytaj_liste_z_csv(plik):
    with open(plik, mode='r', encoding='utf-8') as file:
        reader = csv.reader(file)
        next(reader)  # Pomijanie nagłówka
        return [row for row in reader]

courses = wczytaj_liste_z_csv(courses_csv)
course_modules = wczytaj_liste_z_csv(course_modules_csv)
course_stationary_meetings =
wczytaj_liste_z_csv(course_stationary_meetings_csv)
rooms = wczytaj_liste_z_csv(rooms_csv)

def make_dict_with_min_course_room_capacity():
    min_course_room_capacity = {}
    for course in courses:
        min_course_room_capacity[course[0]] = 1000
    return min_course_room_capacity

min_course_room_capacity = make_dict_with_min_course_room_capacity()

def make_dict_with_room_capacity():
    room_capacity = {}
    for room in rooms:
        room_capacity[room[0]] = int(room[4])
    return room_capacity
room_capacity = make_dict_with_room_capacity()


for meeting in course_stationary_meetings:
    room = meeting[2]
    capacity = room_capacity[room]
    module = meeting[1]
    course = course_modules[int(module)-1][1]
    if capacity < min_course_room_capacity[course]:
        min_course_room_capacity[course] = capacity

for course in courses:
    if min_course_room_capacity[course[0]] == 1000:
        course.pop()
        course.append("")
```

```
    else:
        random_limit = random.randint(18,
min_course_room_capacity[course[0]])
        course.pop()
        course.append(str(random_limit))

with open(output_csv, mode='w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerow(["CourseID", "CourseName", "CourseCoordinatorID",
"CourseDescription", "CoursePrice", "StudentLimit"])
    for course in courses:
        writer.writerow(course)
```

Dodawanie ocen z przedmiotów

```
import random
import csv

# Lista studentów
students = [
9,
12,
26,
27,
31,
36,
37,
39,
44,
67,
90,
92,
105,
108,
114,
130,
142,
144,
163,
171,
179,
180,
189,
```

```
197,
222,
223,
227,
232,
234,
258

]

# Przykładowy zakres przedmiotów (można dostosować według potrzeb)
subjects = [
    205,
    206,
    207,
    208,
    209,
    210,
    211,
    212,
    213,
    214

]  # ID przedmiotów

# Generowanie danych CSV
rows = []

for subject in subjects:
    for student in students:
        grade = random.randint(2, 6)  # Losowanie oceny z zakresu 2-6
        rows.append([subject, student, grade])

# Zapisanie do pliku CSV
with open("grades.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(rows)

print("CSV file 'grades.csv' generated.")
```

Zmiana niektórych kursów na niezdane

```python
import pandas as pd
import numpy as np

# Wczytanie danych z pliku CSV
def modify_csv(input_file, output_file):
    # Wczytaj dane z CSV
    df = pd.read_csv(input_file)

    # Sprawdź kolumnę 'Passed' i znajdź indeksy gdzie jest 1
    ones_indices = df.index[df['Passed'] == 1].tolist()

    # Określ liczbę około 1% jedynek do zmiany
    num_to_change = max(1, int(len(ones_indices) * 0.03))

    # Wybierz losowe indeksy do zamiany
    indices_to_change = np.random.choice(ones_indices, num_to_change,
replace=False)

    # Zmień wybrane jedynki na zera
    df.loc[indices_to_change, 'Passed'] = 0

    # Zapisz zmodyfikowany DataFrame do nowego pliku CSV
    df.to_csv(output_file, index=False)
    print(f"Zmodyfikowano {num_to_change} wartości w kolumnie 'Passed'.
Plik zapisano jako {output_file}.")

# Użycie funkcji
input_file = './CourseModulesPassed.csv'   # Nazwa wejściowego pliku
CSV
output_file = 'zmodyfikowane_dane.csv'  # Nazwa wyjściowego pliku CSV
modify_csv(input_file, output_file)
```

Widoki

vw_UsersWithRoles

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o rolach użytkowników.
- **Zawiera** ID użytkownika (UserID), imię użytkownika (FirstName), nazwisko użytkownika (LastName), email użytkownika (Email), pełnioną przez użytkownika rolę (RoleName)

```
CREATE VIEW vw_UsersWithRoles AS
SELECT
	U.UserID,
	U.FirstName,
	U.LastName,
	U.Email,
	R.RoleName
FROM Users U
	JOIN UsersRoles UR ON U.UserID = UR.UserID
	JOIN Roles R ON UR.RoleID = R.RoleID;
```

---

vw_CoursesWithModules

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o kursach wraz z ich modułami i językiem.
- **Zawiera** ID kursu (CourseID), nazwę kursu (CourseName), opis kursu (CourseDescription), ID modułu kursu (ModuleID), nazwę modułu kursu (ModuleName), Typ modułu kursu (Typ Modułu), nazwę języka, w którym jest prowadzony moduł (ModuleLanguage)

```
CREATE VIEW vw_CoursesWithModules AS
SELECT
	C.CourseID,
	C.CourseName,
	C.CourseDescription,
	CM.ModuleID,
	CM.ModuleName,
	TypeName AS 'Typ Modułu',
	L.LanguageName AS ModuleLanguage
FROM Courses C
JOIN CourseModules CM ON C.CourseID = CM.CourseID
JOIN FormOfActivity ON CM.ModuleType = FormOfActivity.ActivityTypeID
```

```
JOIN Languages L ON CM.LanguageID = L.LanguageID;
```

---

vw_InternshipDetails

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o szczegółach praktyk.
- **Zawiera** ID praktyk (InternshipID), semestr, na którym odbywają się dane praktyki (Term), nazwę kierunku, do którego są przypisane dane praktyki (StudyName), datę rozpoczęcia praktyk (StartDate) i zakończenia praktyk (EndDate), liczbę godzin praktyk (NumberOfHours), imię (FirstName) i nazwisko (LastName) koordynatora

```
CREATE VIEW vw_InternshipDetails AS
SELECT
	I.InternshipID,
	I.Term,
	S.StudyName,
	I.StartDate,
	I.EndDate,
	I.NumerOfHours,
	U.FirstName AS CoordinatorFirstName,
	U.LastName AS CoordinatorLastName
FROM Internship I
	JOIN Employees E ON I.InternshipCoordinatorID = E.EmployeeID
	JOIN Users U ON U.UserID = E.EmployeeID
	JOIN Studies S ON S.StudiesID=I.StudiesID;
```

---

vw_OrdersWithDetail

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o szczegółach zamówień.

Zawiera ID zamówienia (OrderID), datę zamówienia (OrderDate), informację, czy płatność została odroczona (PaymentDeffered), ID

zamówionej aktywności (ActivityID), typ aktywności (ActivityType)

```
CREATE VIEW vw_OrdersWithDetails AS
SELECT
	O.OrderID,
	O.OrderDate,
	O.PaymentDeferred,
	OD.ActivityID,
	AT.TypeName AS ActivityType
FROM Orders O
	JOIN OrderDetails OD ON O.OrderID = OD.OrderID
	JOIN ActivitiesTypes AT ON OD.TypeOfActivity = AT.ActivityTypeID;
```

---

vw_StudyMeetings

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o szczegółach spotkań studyjnych.
- **Zawiera** ID spotkania (MeetingID), nazwę przedmiotu, w ramach którego odbywa się spotkanie (SubjectName), początek (StartTime) i koniec spotkania (EndTime), język spotkania (MeetingLanguage), , imię (FirstName) i nazwisko (LastName) prowadzącego zajęcia

```
CREATE VIEW vw_StudyMeetings AS
SELECT
	SM.MeetingID,
	S.SubjectName,
	SM.StartTime,
	SM.EndTime,
	L.LanguageName AS MeetingLanguage,
	U.FirstName AS LecturerFirstName,
	U.LastName AS LecturerLastName
FROM StudyMeetings SM
	JOIN Subjects S ON SM.SubjectID = S.SubjectID
	JOIN Languages L ON SM.LanguageID = L.LanguageID
	JOIN Users U ON SM.LecturerID = U.UserID;
```

---

vw_SubjectsEachGradesNumber

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o ilości uczniów, którzy zdobyli daną ocenę w ramach przedmiotu.

Zawiera ID przedmiotu (SubjectID), nazwę przedmiotu (SubjectName), ocenę (GradeName) i liczbę uczniów, które ją uzyskało na danym przedmiocie (StudentsNum)

```
CREATE VIEW vw_SubjectsEachGradesNumber AS
SELECT
	S.SubjectID,
	S.SubjectName,
	G.GradeName,
	COUNT(U.UserID) as StudentsNum
FROM SubjectsResults SR
	JOIN Subjects S ON SR.SubjectID = S.SubjectID
	JOIN Users U ON SR.StudentID = U.UserID
	JOIN Grades G ON SR.GradeID = G.GradeID
GROUP BY S.SubjectID,S.SubjectName, G.GradeID, G.GradeName
```

---

vw_EmployeeDegrees

J.K.

OPIS

- **Funkcja**: Przechowuje informacje stopniach naukowych pracowników.

Zawiera ID pracownika (EmployeeID), imię (FirstName) i nazwisko pracownika (LastName), nazwę stopnia naukowego (DegreeName)

```
CREATE VIEW vw_EmployeeDegrees AS
SELECT
	E.EmployeeID,
	U.FirstName,
```

```
	U.LastName,
	D.DegreeName
FROM Employees E
	JOIN Users U ON E.EmployeeID = U.UserID
	JOIN Degrees D ON E.DegreeID = D.DegreeID;
```

---

vw_WebinarsWithDetails

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o webinarach.

Zawiera ID webinaru (WebinarID), nawę webinaru (WebinarName), opis webinaru (WebinarDescription), początek (StartDate), koniec webinaru (EndDate), nazwę języka, w którym webinar jest prowadzony (LanguageName), imię (LectuterFirstName) i nazwisko (LectuterLastName) prowadzącego webinar

```
CREATE VIEW vw_WebinarsWithDetails AS
SELECT
	W.WebinarID,
	W.WebinarName,
	W.WebinarDescription,
	W.StartDate,
	W.EndDate,
	L.LanguageName AS WebinarLanguage,
	U.FirstName AS LecturerFirstName,
	U.LastName AS LecturerLastName
FROM Webinars W
	JOIN Languages L ON W.LanguageID = L.LanguageID
	JOIN Users U ON W.TeacherID = U.UserID;
```

---

vw_InternshipParticipants

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o uczestnikach praktyk.

Zawiera ID praktyk (InternshipID), początek (StartDate), koniec praktyk (EndDate), imię (LectuterFirstName) i nazwisko (LectuterLastName) prowadzącego, informacja, czy zostały zaliczone (Passed)

```
CREATE VIEW vw_InternshipsParticipants AS
SELECT
	I.InternshipID,
	I.StartDate,
	I.EndDate,
	U.FirstName AS StudentFirstName,
	U.LastName AS StudentLastName,
	IIF((IP.Passed = 1), 'Zaliczone', 'Nie zaliczone') AS Passed
FROM Internship I
JOIN InternshipPassed IP ON I.InternshipID = IP.InternshipID
JOIN Users U ON IP.StudentID = U.UserID
JOIN USERS U1 ON I.InternshipCoordinatorID = U1.UserID;
```

---

vw_StudentsGradesWithSubject

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o uzyskanych przez studenta ocenach, w ramach przedmiotów.

Zawiera imię (StudentFirstName) i nazwisko (StudentLastName) studenta, nazwę przedmiotu (SubjectName), ocenę (GradeName)

```
CREATE VIEW vw_StudentsGradesWithSubjects AS
SELECT
	U.FirstName AS StudentFirstName,
	U.LastName AS StudentLastName,
	S.SubjectName,
	G.GradeName
FROM SubjectsResults SR
	JOIN Users U ON SR.StudentID = U.UserID
	JOIN Subjects S ON SR.SubjectID = S.SubjectID
```

```
    JOIN Grades G ON SR.GradeID = G.GradeID;
```

---

vw_LecturerMeetings

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o spotkaniach studyjnych prowadzonych przez użytkownika

Zawiera imię (LecturerFirstName) i nazwisko (LecturerLastName) prowadzącego, nazwę przedmiotu (SubjectName), początek (StartTime) i koniec spotkania (EndTime), język, w którym spotkanie jest prowadzone (MeetingLanguage)

```
CREATE VIEW vw_LecturerMeetings AS
SELECT
    U.FirstName AS LecturerFirstName,
    U.LastName AS LecturerLastName,
    S.SubjectName,
    SM.StartTime,
    SM.EndTime,
    L.LanguageName AS MeetingLanguage
FROM StudyMeetings SM
    JOIN Users U ON SM.LecturerID = U.UserID
    JOIN Subjects S ON SM.SubjectID = S.SubjectID
    JOIN Languages L ON SM.LanguageID = L.LanguageID;
```

---

vw_CourseModulesLanguages

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o językach, w jakich prowadzone są moduły kursów

Zawiera ID kursu (CourseID), nazwę kursu (CourseName), nazwę modułu kursu (ModuleName), język, w którym moduł jest prowadzony

```
(LanguageName)

CREATE VIEW vw_CourseModulesLanguages AS
SELECT
      C.CourseID,
      C.CourseName,
      CM.ModuleName,
      L.LanguageName
FROM Courses C
      JOIN CourseModules CM ON C.CourseID = CM.CourseID
      JOIN Languages L ON CM.LanguageID = L.LanguageID;
```

---

vw_WebinarsLanguages

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o językach, w jakich prowadzone są webinaru

Zawiera ID webinaru (WebinarID), nazwę webinaru (WebinarName), nazwę języku, w którym webinar jest prowadzony (LanguageName)

```
CREATE VIEW vw_WebinarsLanguages AS
SELECT
      W.WebinarID, W.WebinarName, L.LanguageName
FROM Webinars W
      JOIN Languages L ON W.LanguageID = L.LanguageID;
```

---

vw_StudentsDiplomas

K.B.

OPIS

- **Funkcja**: Przechowuje informacje zawarte na dymplomie ukończenia studiów

Zawiera Imię studenta (FirstName), nazwisko studenta (LastName), nazwę kierunku studiów (StudyName), nazwę oceny (GradeName), początek studiów (StudyStart), koniec studiów (StudyEnd)

```
CREATE VIEW vw_StudentsDiplomas AS
SELECT Users.FirstName, Users.LastName, S.StudyName, Grades.GradeName,
      (SELECT CONCAT(DATENAME(MONTH, MIN(StudyMeetings.StartTime)),' ' +
DATENAME(YEAR, MIN(StudyMeetings.StartTime)))
      FROM StudyMeetings
            INNER JOIN Subjects ON
Subjects.SubjectID=StudyMeetings.SubjectID
            INNER JOIN Studies S2 ON S2.StudiesID=Subjects.StudiesID
      WHERE S2.StudiesId=S.StudiesID) as StudyStart,
      (SELECT CONCAT(DATENAME(MONTH, MAX(StudyMeetings.EndTime)),' ' +
DATENAME(YEAR, MAX(StudyMeetings.EndTime)))
      FROM StudyMeetings
            INNER JOIN Subjects ON
Subjects.SubjectID=StudyMeetings.SubjectID
            INNER JOIN Studies S2 ON S2.StudiesID=Subjects.StudiesID
      WHERE S2.StudiesId=S.StudiesID) as StudyEnd
FROM Studies S
      INNER JOIN StudiesResults ON StudiesResults.StudiesID=S.StudiesID
      INNER JOIN Users ON Users.UserID = StudiesResults.StudentID
      INNER JOIN Grades ON StudiesResults.GradeID=Grades.GradeID
```

---

vw_CoursesCertificates

K.B.

OPIS

- **Funkcja**: Przechowuje informacje zawarte na certyfikatach ukończenia kursów

Zawiera Imię użytkownika (FirstName), nazwisko użytkownika (LastName), nazwę kursu (CourseName), początek kursu (CourseStart), koniec kursu (CourseEnd)

```
CREATE VIEW vw_CoursesCertificates AS
SELECT
  Users.FirstName,
  Users.LastName,
  C.CourseName,
  (
      SELECT
      DATENAME(
```

```
      DAY,
      MIN(t.first_meeting_date)
      ) + ' ' + DATENAME(
      MONTH,
      MIN(t.first_meeting_date)
      ) + ' ' + DATENAME(
      YEAR,
      MIN(t.first_meeting_date)
      ) as course_start_date
      FROM
      (
      (
            SELECT
            MIN(
            StationaryCourseMeeting.StartDate
            ) as first_meeting_date
            FROM
            StationaryCourseMeeting
            INNER JOIN CourseModules ON CourseModules.ModuleID =
StationaryCourseMeeting.ModuleID
            INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
            WHERE
            Courses.CourseID = C.CourseID
      )
      UNION
            (
            SELECT
            MIN(OnlineCourseMeeting.StartDate) as first_meeting_date
            FROM
            OnlineCourseMeeting
            INNER JOIN CourseModules ON CourseModules.ModuleID =
OnlineCourseMeeting.ModuleID
            INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
            WHERE
            Courses.CourseID = C.CourseID
            )
      ) as t
  ) as CourseStart,
  (
      SELECT
      (
      DATENAME(
            DAY,
            MAX(t.last_meeting_date)
```

```
      ) + ' ' + DATENAME(
            MONTH,
            MAX(t.last_meeting_date)
      ) + ' ' + DATENAME(
            YEAR,
            MAX(t.last_meeting_date)
      )
      ) as end_date
      FROM
      (
      (
            SELECT
            MAX(
            StationaryCourseMeeting.EndDate
            ) as last_meeting_date
            FROM
            StationaryCourseMeeting
            INNER JOIN CourseModules ON CourseModules.ModuleID =
StationaryCourseMeeting.ModuleID
            INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
            WHERE
            Courses.CourseID = C.CourseID
      )
      UNION
            (
            SELECT
            MAX(OnlineCourseMeeting.EndDate) as last_meeting_date
            FROM
            OnlineCourseMeeting
            INNER JOIN CourseModules ON CourseModules.ModuleID =
OnlineCourseMeeting.ModuleID
            INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
            WHERE
            Courses.CourseID = C.CourseID
            )
      ) as t
  ) as CourseEnd
FROM
  Courses C
  INNER JOIN CourseModules ON CourseModules.CourseID = C.CourseID
  INNER JOIN CourseModulesPassed ON CourseModulesPassed.ModuleID =
CourseModules.ModuleID
  INNER JOIN Users ON Users.UserID = CourseModulesPassed.StudentID
WHERE
```

```
  (
      SELECT
      COUNT(
      DISTINCT CourseModulesPassed.ModuleID
      )
      FROM
      CourseModulesPassed
      INNER JOIN CourseModules ON CourseModules.ModuleID =
CourseModulesPassed.ModuleID
      INNER JOIN Courses C1 ON C1.CourseID = CourseModules.CourseID
      WHERE
      C1.CourseID = C.CourseID
      AND CourseModulesPassed.StudentID = Users.UserID
      AND CourseModulesPassed.Passed = 1
  ) / (
      SELECT
      COUNT(DISTINCT CourseModules.ModuleID)
      FROM
      CourseModules
      INNER JOIN Courses C1 ON C1.CourseID = CourseModules.CourseID
      WHERE
      C1.CourseID = C.CourseID
  ) >= 0.8
```

---

vw_Debtors

K.B.

OPIS

- **Funkcja**: Przechowuje listę dłużników

Zawiera Imię użytkownika (FirstName), nazwisko użytkownika (LastName), email (email), telefon kontaktowy (Phone)

```
CREATE VIEW vw_Debtors AS (
  SELECT
      Users.FirstName,
      Users.LastName,
      Users.Email,
      Users.Phone
  FROM
      Orders
      INNER JOIN Users ON Users.UserID = Orders.StudentID
```

```
      INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
      INNER JOIN StudyMeetingPayment ON StudyMeetingPayment.DetailID =
OrderDetails.DetailID
      INNER JOIN StudyMeetings ON StudyMeetings.MeetingID =
StudyMeetingPayment.MeetingID
  WHERE
      TypeOfActivity =(
      SELECT
      ActivityTypeID
      FROM
      ActivitiesTypes
      WHERE
      TypeName = 'Studia'
      )
      AND (
      OrderDetails.PaidDate is NULL
      OR (
      StudyMeetings.StartTime < DATEADD(
            DAY,
            3,
            GETDATE()
      )
      AND StudyMeetingPayment.PaidDate is NULL
      )
      OR OrderDetails.PaidDate > DATEADD(HOUR, 6, Orders.OrderDate)
      OR (
      StudyMeetingPayment.PaidDate is NOT NULL
      AND StudyMeetingPayment.PaidDate > DATEADD(DAY,-3,
StudyMeetings.StartTime)
      )
      )
)
UNION
  (
      SELECT
      Users.FirstName,
      Users.LastName,
      Users.Email,
      Users.Phone
      FROM
      Orders
      INNER JOIN Users ON Users.UserID = Orders.StudentID
      INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
      INNER JOIN PaymentsAdvances ON PaymentsAdvances.DetailID =
OrderDetails.DetailID
      WHERE
```

```
	TypeOfActivity =(
	SELECT
		ActivityTypeID
	FROM
		ActivitiesTypes
	WHERE
		TypeName = 'Kurs'
	)
	AND (
	(
		AdvancePaidDate is NULL
		AND (
		SELECT
		MIN(t.first_meeting_date)
		FROM
		(
			(
			SELECT
			isnull(
				MIN(
				StationaryCourseMeeting.StartDate
				),
				GETDATE()
			) as first_meeting_date
			FROM
			StationaryCourseMeeting
			INNER JOIN CourseModules ON CourseModules.ModuleID =
StationaryCourseMeeting.ModuleID
			INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
			WHERE
			Courses.CourseID = ActivityID
			)
			UNION
			(
			SELECT
				isnull(
				MIN(OnlineCourseMeeting.StartDate),
				GETDATE()
				) as first_meeting_date
			FROM
				OnlineCourseMeeting
				INNER JOIN CourseModules ON
CourseModules.ModuleID = OnlineCourseMeeting.ModuleID
				INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
```

```
                  WHERE
                        Courses.CourseID = ActivityID
                  )
            ) as t
            )< DATEADD(
            DAY,
            3,
            GETDATE()
            )
      )
      OR PaymentsAdvances.AdvancePaidDate > DATEADD(HOUR, 6,
Orders.OrderDate)
      OR OrderDetails.PaidDate > DATEADD(
            DAY,
            -3,
            (
            SELECT
            MIN(t.first_meeting_date)
            FROM
            (
                  (
                  SELECT
                  isnull(
                        MIN(
                        StationaryCourseMeeting.StartDate
                        ),
                        GETDATE()
                  ) as first_meeting_date
                  FROM
                  StationaryCourseMeeting
                  INNER JOIN CourseModules ON CourseModules.ModuleID =
StationaryCourseMeeting.ModuleID
                  INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
                  WHERE
                  Courses.CourseID = ActivityID
                  )
                  UNION
                  (
                  SELECT
                        isnull(
                        MIN(OnlineCourseMeeting.StartDate),
                        GETDATE()
                        ) as first_meeting_date
                  FROM
                        OnlineCourseMeeting
```

```
                        INNER JOIN CourseModules ON
CourseModules.ModuleID = OnlineCourseMeeting.ModuleID
                        INNER JOIN Courses ON Courses.CourseID =
CourseModules.CourseID
                  WHERE
                        Courses.CourseID = ActivityID
                  )
            ) as t
            )
      )
      )
  )
UNION
  (
      SELECT
      Users.FirstName,
      Users.LastName,
      Users.Email,
      Users.Phone
      FROM
      Orders
      INNER JOIN Users ON Users.UserID = Orders.StudentID
      INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
      INNER JOIN Webinars ON Webinars.WebinarID = OrderDetails.ActivityID
      WHERE
      TypeOfActivity =(
      SELECT
            activityTypeID
      FROM
            ActivitiesTypes
      WHERE
            TypeName = 'Webinar'
      )
      AND (Webinars.Price IS NOT NULL)
      AND (
      PaidDate is NULL
      OR PaidDate > Webinars.StartDate
      )
  )
UNION
  (
      SELECT
      Users.FirstName,
      Users.LastName,
      Users.Email,
      Users.Phone
```

```
	FROM
	Orders
	INNER JOIN Users ON Users.UserID = Orders.StudentID
	INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
	INNER JOIN StudyMeetings ON StudyMeetings.MeetingID = ActivityId
	WHERE
	TypeOfActivity =(
	SELECT
		ActivityTypeID
	FROM
		ActivitiesTypes
	WHERE
		TypeName = 'Spotkanie studyjne'
	)
	AND (
	(
		StudyMeetings.StartTime < DATEADD(
		DAY,
		3,
		GETDATE()
		)
		AND OrderDetails.PaidDate is NULL
	)
	OR (
		OrderDetails.PaidDate is NOT NULL
		AND OrderDetails.PaidDate > DATEADD(DAY,-3,
StudyMeetings.StartTime)
	)
	)
  )
```

VW_IncomeFromWebinars

O.B.

OPIS

- **Funkcja**: Przechowuje infomacje o przychodach w webinarów.
- **Zawiera** ID webinaru (WebinarID), nazwę webinaru (WebinarName), przychód z danego webinaru (Przychód)

```
CREATE VIEW VW_IncomeFromWebinars AS
SELECT WebinarID, WebinarName, ROUND(SUM(OrderDetails.Price), 2) as
```

```
'Przychód'
FROM Webinars
         INNER JOIN OrderDetails ON ActivityID = WebinarID
WHERE TypeOfActivity = 1 AND PaymentStatus = 'udana'
GROUP BY WebinarID, WebinarName
```

VW_IncomeFromCourses

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o przychodach z kursów.
- **Zawiera** ID kursu (CourseID), nazwę kursu (CourseName), przychód z danego kursu (Przychód).

```
CREATE VIEW VW_IncomeFromCourses AS
SELECT CourseID, CourseName, ROUND(SUM(OrderDetails.Price), 2) as
'Przychód'
FROM Courses
         INNER JOIN OrderDetails ON ActivityID = CourseID
    AND TypeOfActivity = 2 AND PaymentStatus = 'udana'
         INNER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
GROUP BY CourseID, CourseName
```

VW_IncomeFromStudies

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o przychodach ze studiów.
- **Zawiera** ID studiów (StudiesID), nazwę studiów (StudyName), przychód z danych studiów (Przychód).

```
CREATE VIEW VW_IncomeFromStudies AS
SELECT StudiesID, StudyName, (
    (SELECT SUM(Price) FROM OrderDetails
     WHERE ActivityID = StudiesID AND TypeOfActivity = 3 AND
PaymentStatus = 'udana'
     GROUP BY ActivityID
    )
```

```
        +
    (SELECT SUM(Price) FROM StudyMeetingPayment
                                INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID = StudyMeetingPayment.MeetingID
                                INNER JOIN Subjects ON
StudyMeetings.SubjectID = Subjects.SubjectID
     WHERE Subjects.StudiesID = Studies.StudiesID AND PaymentStatus =
'udana'
     GROUP BY Subjects.StudiesID
    )
        +
    (SELECT ISNULL(SUM(Price), 0) FROM Subjects
                                            LEFT OUTER JOIN StudyMeetings
ON Subjects.SubjectID = StudyMeetings.SubjectID
                                            LEFT OUTER JOIN OrderDetails
ON OrderDetails.ActivityID = StudyMeetings.MeetingID
        AND TypeOfActivity = 4 AND PaymentStatus = 'udana'
     WHERE Subjects.StudiesID = Studies.StudiesID
     GROUP BY Subjects.StudiesID
    )
    ) AS 'Income'
FROM Studies
```

VW_IncomeFromAllFormOfEducation

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o przychodach z różnych form edukacji.
- **Zawiera** Typ edukacji (Typ), ID (WebinarID/CourseID/StudiesID), nazwę (WebinarName/CourseName/StudyName), przychód (Przychód).

```
CREATE VIEW VW_IncomeFromAllFormOfEducation AS
SELECT 'Webinar' AS 'Typ', WebinarID, WebinarName,
ROUND(SUM(OrderDetails.Price), 2) as 'Przychód'
FROM Webinars
         INNER JOIN OrderDetails ON ActivityID = WebinarID
    AND TypeOfActivity = 1 AND PaymentStatus = 'udana'
         INNER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
GROUP BY WebinarID, WebinarName
UNION
SELECT 'Kurs' AS 'Typ', CourseID, CourseName,
```

```
ROUND(SUM(OrderDetails.Price), 2) as 'Przychód'
FROM Courses
          INNER JOIN OrderDetails ON ActivityID = CourseID
    AND TypeOfActivity = 2 AND PaymentStatus = 'udana'
          INNER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
GROUP BY CourseID, CourseName
UNION
SELECT 'Studia' AS 'Typ',  StudiesID, StudyName, (
    (SELECT ISNULL(SUM(Price), 0) FROM OrderDetails
     WHERE ActivityID = StudiesID AND TypeOfActivity = 3 AND
PaymentStatus = 'udana'
     GROUP BY ActivityID
    )
    +
    (SELECT ISNULL(SUM(Price), 0) FROM StudyMeetingPayment
    INNER JOIN StudyMeetings ON StudyMeetings.MeetingID =
StudyMeetingPayment.MeetingID
    INNER JOIN Subjects ON StudyMeetings.SubjectID = Subjects.SubjectID
     WHERE Subjects.StudiesID = Studies.StudiesID AND PaymentStatus =
'udana'
     GROUP BY Subjects.StudiesID
    )
    +
    (SELECT ISNULL(SUM(Price), 0) FROM Subjects
        LEFT OUTER JOIN StudyMeetings ON Subjects.SubjectID =
StudyMeetings.SubjectID
        LEFT OUTER JOIN OrderDetails ON OrderDetails.ActivityID =
StudyMeetings.MeetingID
        AND TypeOfActivity = 4 AND PaymentStatus = 'udana'
     WHERE Subjects.StudiesID = Studies.StudiesID
     GROUP BY Subjects.StudiesID
    )
    ) AS 'Income'
FROM Studies
```

VW_RoomsAvailability

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o dostępności sal.
- **Zawiera** ID sali (RoomID), nazwę sali (RoomName), datę rozpoczęcia lub godzinę rozpoczęcia (StartDate/StartTime), datę zakończenia lub

godzinę zakończenia (EndDate/EndTime).

```
CREATE VIEW VW_RoomsAvailability AS
SELECT Rooms.RoomID, RoomName, StartDate, EndDate FROM Rooms
INNER JOIN StationaryCourseMeeting ON Rooms.RoomID =
StationaryCourseMeeting.RoomID
UNION
SELECT Rooms.RoomID, RoomName, StartTime, EndTime FROM Rooms
INNER JOIN StationaryMeetings ON Rooms.RoomID =
StationaryMeetings.RoomID
INNER JOIN StudyMeetings ON StationaryMeetings.MeetingID =
StudyMeetings.MeetingID
```

VW_NumberOfStudentsSignUpForFutureWebinars

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o liczbie studentów zapisanych na przyszłe webinary.
- **Zawiera** ID webinaru (WebinarID), nazwę webinaru (WebinarName), Typ aktywności (Typ), liczbę uczestników ('Liczba uczestników').

```
CREATE VIEW VW_NumberOfStudentsSignUpForFutureWebinars AS
SELECT WebinarID, WebinarName, 'Zdalny' as 'Typ', COUNT(StudentID) as
'Liczba uczestników' FROM Webinars
LEFT OUTER JOIN OrderDetails ON OrderDetails.ActivityID =
Webinars.WebinarID
AND TypeOfActivity = 1
LEFT OUTER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
WHERE StartDate > '2025-01-01 00:00'
GROUP BY WebinarID, WebinarName
```

VW_NumberOfStudentsSignUpForFutureCourses

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o liczbie studentów zapisanych na przyszłe kursy oraz typ kursu (zdalny, hybrydowy lub stacjonarny).

- **Zawiera** ID kursu (CourseID), nazwę kursu (CourseName), typ kursu (Typ), liczbę uczestników (Liczba uczestników).

```
CREATE VIEW VW_NumberOfStudentsSignUpForFutureCourses AS
SELECT CourseID, CourseName,
       CASE
           WHEN CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType = 4) THEN 'Hybrydowy'
           WHEN CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType IN (2,3))
               AND
                CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType = 1) THEN 'Hybrydowy'
           WHEN CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType IN (2,3)) THEN 'Zdalny'
           ELSE 'Stacjonarny'
           END AS 'Typ',
       COUNT(StudentID) AS 'Liczba osób'
FROM Courses
         LEFT OUTER JOIN OrderDetails ON OrderDetails.ActivityID =
Courses.CourseID
    AND TypeOfActivity = 2
         LEFT OUTER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
WHERE CourseID IN (
    SELECT DISTINCT Courses.CourseID FROM Courses
                                              INNER JOIN CourseModules
ON Courses.CourseID = CourseModules.CourseID
    WHERE ModuleID IN (
        SELECT ModuleID FROM StationaryCourseMeeting WHERE StartDate >
'2025-01-01'
        UNION
        SELECT ModuleID FROM OnlineCourseMeeting WHERE StartDate >
'2025-01-01'
```

```
    ))
GROUP BY CourseID, CourseName
```

VW_NumberOfStudentsSignUpForFutureStudyMeetings

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o liczbie studentów zapisanych na przyszłe spotkania studyjne.
- **Zawiera** ID spotkania (MeetingID), nazwę spotkania (Nazwa spotkania), liczbę uczestników (Liczba uczestników).

```
CREATE VIEW VW_NumberOfStudentsSignUpForFutureStudyMeetings AS
SELECT MeetingID, 'Spotkanie studyjne ' + STR(MeetingID) AS 'Nazwa
spotkania',
        (
        (SELECT COUNT(*) FROM StudyMeetingPayment WHERE PaymentStatus =
'udana' AND StudyMeetingPayment.MeetingID = StudyMeetings.MeetingID)
            +
        (SELECT COUNT(*) FROM OrderDetails WHERE PaymentStatus = 'udana'
AND TypeOfActivity = 4 AND ActivityID = StudyMeetings.MeetingID)
        ) AS 'Liczba uczestników'
FROM StudyMeetings
WHERE StartTime > '2025-01-01'
```

VW_NumberOfStudentsSignUpForFutureAllFormOfEducation

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o liczbie studentów zapisanych na przyszłe formy edukacji (webinary, kursy i spotkania studyjne).
- **Zawiera** ID aktywności (WebinarID/CourseID/MeetingID), nazwę aktywności (Nazwa), typ aktywności (Typ), liczbę uczestników (Liczba uczestników).

```
CREATE VIEW VW_NumberOfStudentsSignUpForFutureAllFormOfEducation AS
SELECT WebinarID, 'Webinar ' + WebinarName AS 'Nazwa Webinaru', 'Zdalny'
as 'Typ', COUNT(StudentID) as 'Liczba uczestników' FROM Webinars
```

```
LEFT OUTER JOIN OrderDetails ON OrderDetails.ActivityID =
Webinars.WebinarID
AND TypeOfActivity = 1
LEFT OUTER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
WHERE StartDate > '2025-01-01 00:00'
GROUP BY WebinarID, WebinarName
UNION
SELECT CourseID, 'Kurs ' + CourseName,
       CASE
           WHEN CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType = 4) THEN 'Hybrydowy'
           WHEN CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType IN (2,3))
               AND
                CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType = 1) THEN 'Hybrydowy'
           WHEN CourseID IN (SELECT Courses.CourseID
                             FROM Courses
                                      INNER JOIN CourseModules ON
Courses.CourseID = CourseModules.CourseID
                             WHERE ModuleType IN (2,3)) THEN 'Zdalny'
           ELSE 'Stacjonarny'
           END,
       COUNT(StudentID)
FROM Courses
         LEFT OUTER JOIN OrderDetails ON OrderDetails.ActivityID =
Courses.CourseID
    AND TypeOfActivity = 2
         LEFT OUTER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
WHERE CourseID IN (
    SELECT DISTINCT Courses.CourseID FROM Courses
                                              INNER JOIN CourseModules
ON Courses.CourseID = CourseModules.CourseID
    WHERE ModuleID IN (
        SELECT ModuleID FROM StationaryCourseMeeting WHERE StartDate >
'2025-01-01'
        UNION
```

```
        SELECT ModuleID FROM OnlineCourseMeeting WHERE StartDate >
'2025-01-01'
    ))
GROUP BY CourseID, CourseName
UNION
SELECT MeetingID, 'Spotkanie studyjne ' + STR(MeetingID),
IIF(MeetingType = 1, 'Stacjonarne', 'Zdalne'),
        (
        (SELECT COUNT(*) FROM StudyMeetingPayment WHERE PaymentStatus =
'udana' AND StudyMeetingPayment.MeetingID = StudyMeetings.MeetingID)
            +
        (SELECT COUNT(*) FROM OrderDetails WHERE PaymentStatus = 'udana'
AND TypeOfActivity = 4 AND ActivityID = StudyMeetings.MeetingID)
        )
FROM StudyMeetings
WHERE StartTime > '2025-01-01'
```

VW_StudyMeetingsPresenceWithFirstNameLastNameDate

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o obecności studentów na spotkaniach studyjnych, w tym imię, nazwisko oraz status obecności.
- **Zawiera** ID spotkania (MeetingID), datę i godzinę spotkania (StartTime), imię (FirstName), nazwisko (LastName), status obecności (Obecność).

```
CREATE VIEW VW_StudyMeetingsPresenceWithFirstNameLastNameDate AS
SELECT MeetingID, StartTime, FirstName, LastName, IIF(Presence = 1,
'Obecny', 'Nieobecny') AS 'Obecność' FROM StudyMeetingPresence
INNER JOIN StudyMeetings ON StudyMeetingPresence.StudyMeetingID =
StudyMeetings.MeetingID
INNER JOIN Users ON StudyMeetingPresence.StudentID = Users.UserID
```

VW_LanguagesAndTranslatorsOnWebinars

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o językach i tłumaczach

przypisanych do webinarów.
- **Zawiera** ID webinaru (WebinarID), nazwę webinaru (WebinarName), nazwę języka (LanguageName), imię tłumacza (FirstName), nazwisko tłumacza (LastName). Jeśli tłumacz nie jest przypisany, wyświetli się 'brak'.

```
CREATE VIEW VW_LanguagesAndTranslatorsOnWebinars AS
SELECT WebinarID, WebinarName, LanguageName, ISNULL(FirstName, 'brak')
AS 'FirstName', ISNULL(LastName, 'brak') AS 'LastName' FROM Webinars
INNER JOIN Languages ON Webinars.LanguageID = Languages.LanguageID
LEFT OUTER JOIN Employees ON Webinars.TranslatorID =
Employees.EmployeeID
LEFT OUTER JOIN Users ON Employees.EmployeeID = Users.UserID
```

VW_LanguagesAndTranslatorsOnCourses

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o językach i tłumaczach przypisanych do kursów.
- **Zawiera** ID kursu (CourseID), nazwę kursu (CourseName), nazwę języka (LanguageName), imię tłumacza (FirstName), nazwisko tłumacza (LastName). Jeśli tłumacz nie jest przypisany, wyświetli się 'brak'.

```
CREATE VIEW VW_LanguagesAndTranslatorsOnCourses AS
SELECT Courses.CourseID, CourseName, LanguageName, ISNULL(FirstName,
'brak') AS 'FirstName', ISNULL(LastName, 'brak') AS 'LastName' FROM
Courses
INNER JOIN CourseModules ON Courses.CourseID = CourseModules.CourseID
INNER JOIN Languages ON CourseModules.LanguageID = Languages.LanguageID
LEFT OUTER JOIN Employees ON CourseModules.TranslatorID =
Employees.EmployeeID
LEFT OUTER JOIN Users ON Employees.EmployeeID = Users.UserID
```

VW_LanguagesAndTranslatorsOnStudies

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o językach i tłumaczach przypisanych do spotkań studyjnych, wraz z nazwą studiów, przedmiotów oraz ID spotkania.
- **Zawiera** ID studiów (StudiesID), nazwę studiów (StudyName), nazwę przedmiotu (SubjectName), ID spotkania (MeetingID), nazwę języka (LanguageName), imię tłumacza (FirstName), nazwisko tłumacza (LastName). Jeśli tłumacz nie jest przypisany, wyświetli się 'brak'.

```
CREATE VIEW VW_LanguagesAndTranslatorsOnStudies AS
SELECT Studies.StudiesID, StudyName, SubjectName, MeetingID,
LanguageName, ISNULL(FirstName, 'brak') AS 'FirstName', ISNULL(LastName,
'brak') AS 'LastName' FROM Studies
INNER JOIN Subjects ON Studies.StudiesID = Subjects.StudiesID
INNER JOIN StudyMeetings ON Subjects.SubjectID = StudyMeetings.SubjectID
INNER JOIN Languages ON StudyMeetings.LanguageID = Languages.LanguageID
LEFT OUTER JOIN Employees ON StudyMeetings.TranslatorID =
Employees.EmployeeID
LEFT OUTER JOIN Users ON Employees.EmployeeID = Users.UserID
```

VW_StudentsPlaceOfLive

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o miejscu zamieszkania studentów, w tym imię, nazwisko oraz miasto.
- **Zawiera:** ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę miasta (CityName).

```
CREATE VIEW VW_StudentsPlaceOfLive AS
SELECT Users.UserID, FirstName, LastName, CityName FROM Users
INNER JOIN UsersRoles ON Users.UserID = UsersRoles.UserID
AND RoleID = 1
INNER JOIN Cities ON Users.CityID = Cities.CityID
```

VW_StudiesCoordinators

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o koordynatorach studiów, w tym nazwę studiów oraz imię i nazwisko koordynatora.
- **Zawiera:** ID studiów (StudiesID), nazwę studiów (StudyName), imię koordynatora (FirstName), nazwisko koordynatora (LastName).

```
CREATE VIEW VW_StudiesCoordinators AS
SELECT StudiesID, StudyName, FirstName, LastName FROM Studies
INNER JOIN Employees ON Studies.StudiesCoordinatorID =
Employees.EmployeeID
INNER JOIN Users ON Employees.EmployeeID = Users.UserID
```

VW_CoursesCoordinators

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o koordynatorach kursów, w tym nazwę kursu oraz imię i nazwisko koordynatora.
- **Zawiera:** ID kursu (CourseID), nazwę kursu (CourseName), imię koordynatora (FirstName), nazwisko koordynatora (LastName).

```
CREATE VIEW VW_CoursesCoordinators AS
SELECT CourseID, CourseName, FirstName, LastName FROM Courses
INNER JOIN Employees ON Courses.CourseCoordinatorID =
Employees.EmployeeID
INNER JOIN Users ON Employees.EmployeeID = Users.UserID
```

VW_CoursesStartDateEndDate

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o dacie rozpoczęcia i zakończenia kursów, uwzględniając zarówno spotkania stacjonarne, jak i online.
- **Zawiera:** ID kursu (ID), datę rozpoczęcia kursu (Data rozpoczęcia), datę zakończenia kursu (Data zakończenia).

```
CREATE VIEW VW_CoursesStartDateEndDate AS
```

```
WITH T1 AS (
    SELECT
        Courses.CourseID AS 'ID',
        MIN(OnlineCourseMeeting.StartDate) AS 'online_start_date',
        MIN(StationaryCourseMeeting.StartDate) AS
'stationary_start_date',
        MAX(OnlineCourseMeeting.EndDate) AS 'online_end_date',
        MAX(StationaryCourseMeeting.EndDate) AS 'stationary_end_date'
    FROM
        Courses
            INNER JOIN
        CourseModules ON Courses.CourseID = CourseModules.CourseID
            LEFT OUTER JOIN
        StationaryCourseMeeting ON CourseModules.ModuleID =
StationaryCourseMeeting.ModuleID
            LEFT OUTER JOIN
        OnlineCourseMeeting ON CourseModules.ModuleID =
OnlineCourseMeeting.ModuleID
    GROUP BY
        Courses.CourseID
)
SELECT
    id,
    CASE
        WHEN COALESCE(online_start_date, '2050-12-31') <
COALESCE(stationary_start_date, '2050-12-31')
            THEN COALESCE(online_start_date, stationary_start_date)
        ELSE COALESCE(stationary_start_date, online_start_date)
        END AS 'Data rozpoczęcia',

    CASE
        WHEN COALESCE(online_end_date, '2010-01-01') >
COALESCE(stationary_end_date, '2010-01-01')
            THEN COALESCE(online_end_date, stationary_end_date)
        ELSE COALESCE(stationary_end_date, online_end_date)
        END AS 'Data zakończenia'
FROM
    T1;
```

VW_EmployeesFunctionsAndSeniority

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o pracownikach, ich funkcjach oraz stażu pracy (w latach, miesiącach i dniach).
- **Zawiera:** ID pracownika (EmployeeID), imię (FirstName), nazwisko (LastName), nazwę roli (RoleName), oraz staż pracy (RóżnicaCzasu) w formie "X lat Y miesięcy Z dni".

```
CREATE VIEW VW_EmployeesFunctionsAndSeniority AS
SELECT EmployeeID, FirstName, LastName, RoleName, CONCAT(
        DATEDIFF(YEAR, HireDate, '2025-01-01'), ' lat ',
        DATEDIFF(MONTH, DATEADD(YEAR, DATEDIFF(YEAR, '2025-01-01',
HireDate), '2025-01-01'), HireDate), ' miesiące ',
        DATEDIFF(DAY, DATEADD(MONTH, DATEDIFF(MONTH, DATEADD(YEAR,
DATEDIFF(YEAR, '2025-01-01', HireDate), '2025-01-01'), HireDate),
                                DATEADD(YEAR, DATEDIFF(YEAR, '2025-01-01',
HireDate), '2025-01-01')), HireDate), ' dni'
                                                        ) AS RóżnicaCzasu
FROM Employees
         INNER JOIN Users ON Employees.EmployeeID = Users.UserID
         INNER JOIN UsersRoles ON Users.UserID = UsersRoles.UserID
         INNER JOIN Roles ON UsersRoles.RoleID = Roles.RoleID
```

VW_StudyMeetingDurationTimeNumberOfStudents

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o czasie trwania spotkań studyjnych oraz liczbie zapisanych uczestników.
- **Zawiera:** ID spotkania (MeetingID), nazwę przedmiotu (SubjectName), czas trwania spotkania (Czas trwania) w minutach oraz liczbę zapisanych osób (Liczba zapisanych osób).

```
CREATE VIEW VW_StudyMeetingDurationTimeNumberOfStudents AS
SELECT MeetingID, SubjectName, STR(DATEDIFF(minute, StartTime, EndTime))
+ ' minut' AS 'Czas trwania',
       (
           (SELECT COUNT(*) FROM StudyMeetingPayment WHERE PaymentStatus
= 'udana' AND StudyMeetingPayment.MeetingID = StudyMeetings.MeetingID)
               +
           (SELECT COUNT(*) FROM OrderDetails WHERE PaymentStatus =
```

```
'udana' AND TypeOfActivity = 4 AND ActivityID = StudyMeetings.MeetingID)
           ) AS 'Liczba zapisanych osób'
FROM StudyMeetings
         INNER JOIN Subjects ON StudyMeetings.SubjectID =
Subjects.SubjectID
```

VW_UsersPersonalData

J.K.

OPIS

- **Funkcja**: Przechowuje informację o danych personalnych użytkowników.
- **Zawiera:** imię użytkownika (FirstName), nazwisko użytkownika (LastName), email (email), ulica i numer (Street), kod pocztowy (PostalCode), nazwę miasta (CityName)

```
CREATE VIEW VW_UsersPersonalData AS
SELECT FirstName, LastName, Email, Phone, Street, PostalCode, CityName
FROM Users
         INNER JOIN Cities ON Users.CityID = Cities.CityID
```

VW_UsersDiplomasAddresses

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o danych do wysyłki dyplomów studiów
- **Zawiera:** imię użytkownika (FirstName), nazwisko użytkownika (FirstName), miasto i numer (Street), kod pocztowy (PostalCode), nazwę miasta (CityName)

```
CREATE VIEW VW_UsersDiplomasAddresses AS
SELECT FirstName, LastName, Street, PostalCode, CityName
FROM Studies S
         INNER JOIN StudiesResults ON
StudiesResults.StudiesID=S.StudiesID
         INNER JOIN Users ON Users.UserID = StudiesResults.StudentID
         INNER JOIN Grades ON StudiesResults.GradeID=Grades.GradeID
```

```
        INNER JOIN Cities ON Cities.CityID = Users.CityID
```

---

vw_CoursesCertificatesAddresses

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o danych do wysyłki certyfikatów ukończenia kursu
- **Zawiera:** imię użytkownika (FirstName), nazwisko użytkownika (FirstName), miasto i numer (Street), kod pocztowy (PostalCode), nazwę miasta (CityName)

```
CREATE VIEW vw_CoursesCertificatesAddresses AS
SELECT Users.FirstName, Users.LastName, Street, PostalCode, CityName
FROM Courses C
         INNER JOIN CourseModules ON CourseModules.CourseID=C.CourseID
         INNER JOIN CourseModulesPassed ON
CourseModulesPassed.ModuleID=CourseModules.ModuleID
         INNER JOIN Users ON Users.UserID =
CourseModulesPassed.StudentID
         INNER JOIN Cities ON Cities.CityID = Users.CityID
WHERE (
          SELECT COUNT(DISTINCT CourseModulesPassed.ModuleID)
          FROM CourseModulesPassed
                   INNER JOIN CourseModules ON
CourseModules.ModuleID=CourseModulesPassed.ModuleID
                   INNER JOIN Courses C1 ON
C1.CourseID=CourseModules.CourseID
          WHERE C1.CourseID=C.CourseID AND
CourseModulesPassed.StudentID=Users.UserID AND
CourseModulesPassed.Passed=1)
          /
      (SELECT COUNT(DISTINCT CourseModules.ModuleID)
       FROM CourseModules
                INNER JOIN Courses C1 ON
C1.CourseID=CourseModules.CourseID
       WHERE C1.CourseID=C.CourseID) >= 0.8
```

---

vw_PresenceOnPastStudyMeetings

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o frekwencji (procent obecnych osób) na przeszłym spotkaniu studyjnym.
- **Zawiera** ID spotkania studyjnego (MeetingID), Procent obecnych osób na danych zajęciach (PercentOfPresentStudents)

```
CREATE VIEW vw_ AS PresenceOnPastStudyMeeting
WITH t2 AS (
    SELECT
        sm.MeetingID,
        COUNT(smp.Presence) AS NumberOfAttendees,
        (
            SELECT
                COUNT(DISTINCT od.OrderID) AS NumberOfPurchases
            FROM
                StudyMeetings sm1
            LEFT JOIN
                Subjects s ON s.SubjectID = sm1.SubjectID
            LEFT JOIN
                Studies st ON s.StudiesID = st.StudiesID
            LEFT JOIN
                OrderDetails od ON (od.ActivityID = sm1.MeetingID AND
od.TypeOfActivity = 4)
                                OR (od.ActivityID = st.StudiesID AND
od.TypeOfActivity = 3)
            WHERE sm1.MeetingID = sm.MeetingID
            GROUP BY
                sm1.MeetingID
        ) AS NumberOfPurchases
    FROM
        StudyMeetings sm
    LEFT JOIN
        StudyMeetingPresence smp ON smp.StudyMeetingID = sm.MeetingID
AND smp.Presence = 1
    WHERE EndTime < '2025-01-01'
    GROUP BY
        sm.MeetingID
)

SELECT
    t2.MeetingID,ROUND((CAST(NumberOfAttendees AS FLOAT) /
CAST(NumberOfPurchases AS FLOAT)) * 100, 2) AS PercentOfPresentStudents
FROM
    t2;
```

```
CREATE VIEW vw_StudentsAttendanceAtSubjects AS
SELECT
    smp.StudentID,
    sm.SubjectID,
    CAST (SUM(CAST(smp.Presence AS INT)) * 1.0 / COUNT(*) * 100 AS INT)
AS AttendancePercentage
FROM
    StudyMeetingPresence smp
        INNER JOIN
    StudyMeetings sm ON sm.MeetingID = smp.StudyMeetingID
GROUP BY
    smp.StudentID,
    sm.SubjectID;
```

---

vw_StudentsAttendanceAtSubjects

J.K.

OPIS

- **Funkcja**: Przechowuje informacje o frekwencji danego studenta na przypisanych przedmiotach
- **Zawiera** ID studenta (StudentID), ID przedmiotu (SubjectID), frekwencję studenta na danym przedmiocie (procent jego obecności) (AttendancePercentage)

```
CREATE VIEW vw_StudentsAttendanceAtSubjects AS
SELECT
    smp.StudentID,
    sm.SubjectID,
    CAST (SUM(CAST(smp.Presence AS INT)) * 1.0 / COUNT(*) * 100 AS INT) AS
AttendancePercentage
FROM
    StudyMeetingPresence smp
INNER JOIN
    StudyMeetings sm ON sm.MeetingID = smp.StudyMeetingID
GROUP BY
    smp.StudentID,
    sm.SubjectID
ORDER BY
    AttendancePercentage;
```

---

vw_bilocationReport

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o przyszłych spotkaniach studyjnych, na które użytkownik jest zapisany a które kolidują czasowo
- **Zawiera:** ID studenta (StudentID),ID pierwszego kolidującego spotkania (first_meeting_id), Datę rozpoczęcia pierwszego spotkania (first_meeting_start_time), Datę zakończenia pierwszego spotkania (first_meeting_end_time), ID drugiego kolidującego spotkania (second_meeting_id), Datę rozpoczęcia drugiego spotkania (second_meeting_start_time), datę zakończenia drugiego spotkania (second_meeting_end_time)

```
CREATE VIEW vw_bilocationReport AS
 WITH student_meetings AS (
 SELECT
 users.userId,
 orderdetails.ActivityID AS meetingId,
 StudyMeetings.startTime,
 StudyMeetings.endTime
 FROM Users
 JOIN orders on orders.StudentID=users.UserID
 JOIN OrderDetails on OrderDetails.OrderID = orders.orderid
 JOIN StudyMeetings on StudyMeetings.MeetingID=OrderDetails.ActivityID
 where TypeOfActivity = 3
 )
 SELECT
 student_meetings.userId,
 student_meetings.meetingid AS first_meeting_id,
 student_meetings.starttime AS first_meeting_start_time,
 student_meetings.endtime AS first_meeting_end_time,
 SM.meetingid AS second_meeting_id,
 SM.starttime AS second_meeting_start_time,
 SM.endtime AS second_meeting_end_time
 FROM student_meetings
 JOIN orders O ON student_meetings.userid = O.studentid
 JOIN OrderDetails od on od.orderid=O.orderid
 JOiN StudyMeetingPayment SMP on SMP.detailid= od.detailid
 JOIN StudyMeetings SM on SMP.meetingid = SM.meetingid
 WHERE student_meetings.meetingid < SM.meetingid AND
 (student_meetings.starttime >= GETDATE() OR SM.startTime >=
 GETDATE()) AND
 (student_meetings.endtime >= SM.startTime AND
```

```
 student_meetings.starttime <= SM.endTime)
```

---

vw_defferedPayments

K.B.

OPIS

- **Funkcja**: Przechowuje zamówienia z odroczoną płatnością
- **Zawiera:** imię użytkownika (OrderID), ID użytkownika (StudentID), ID aktywności (ActivityID), typ aktywności (TypeName), DefferedDate (DefferedDate),

```
CREATE VIEW defferedPayments AS
 SELECT orders.orderid, orders.StudentID, orderdetails.activityid,
(select typename FROM FormOfActivity where
ActivityTypeID=OrderDetails.TypeOfActivity) AS 'Typ', DeferredDate
 FROM orders
 INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
 WHERE Orders.PaymentDeferred=1
```

---

vw_MeetingsWithAbsences

J.K.

OPIS

- **Funkcja**: Przechowuje wszystkie spotkania oraz studentów, podczas których dany student był nieobecny z informacją, czy zajęcia te zostały odrobione
- **Zawiera:** id spotkania na którym student był nieobecny (StudyMeetingID), ID tego studenta (StudentID), Informację o obecności na przeszłym spotkaniu (Presence), informację o obecności na innej aktywności (OtherActivityPresence)

```
CREATE VIEW vw_ MeetingsWithAbsences AS
SELECT
  smp.StudyMeetingID,
  smp.StudentID,
  smp.Presence,
  1 as OtherActivityPresence
FROM StudyMeetingPresence smp
INNER JOIN ActivityInsteadOfAbsence aioa
```

```
  ON aioa.MeetingID = smp.StudyMeetingID
  AND aioa.StudentID = smp.StudentID

UNION

SELECT
  smp.StudyMeetingID,
  smp.StudentID,
  smp.Presence,
  0 as OtherActivityPresence
FROM StudyMeetingPresence smp
LEFT OUTER JOIN ActivityInsteadOfAbsence aioa
  ON aioa.MeetingID = smp.StudyMeetingID
  AND aioa.StudentID = smp.StudentID
WHERE
  smp.presence = 0
  AND aioa.MeetingID IS NULL;
```

VW_allUsersCourseMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje wszystkie kursy na które dany użytkownik był zapisany
- **Zawiera:** id użytkownika (StudentID), imię tego studenta (FirstName), nazwisko tego studenta (LastName), nazwa kursu (NazwaKursu), nazwa modułu kursu (NazwaModulu), data początku kursy (StartDate), data końca kursu (EndDate), status płatności (PaymentStatus)

```
CREATE VIEW VW_allUsersCourseMeetings AS
SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    StationaryCourseMeeting.StartDate,
    StationaryCourseMeeting.EndDate,
    OrderDetails.PaymentStatus
```

```
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    OnlineCourseMeeting.StartDate,
    OnlineCourseMeeting.EndDate,
    OrderDetails.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID;
```

VW_BilocationBetweenAllActivities

O.B.

OPIS

- **Funkcja**: Przechowuje wszystkie spotkania studyjne
- **Zawiera:** id użytkownika (StudentID), imię tego studenta (FirstName), nazwisko tego studenta (LastName), nazwa kursu (NazwaKursu), nazwa modułu kursu (NazwaModulu), data początku kursy (StartDate), data końca kursu (EndDate), status płatności (PaymentStatus)

```
CREATE VIEW VW_BilocationBetweenAllActivities AS
WITH T1 (StudentID, StartTime, EndTime, TypeOfActivity, ActivityID) AS (
        SELECT StudentID, StartDate, EndDate, 'Webinar', WebinarID FROM
Orders
        INNER JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
```

```
        AND TypeOfActivity = 1
        INNER JOIN Webinars ON WebinarID = OrderDetails.ActivityID
        WHERE StartDate > '2025-01-01'
        UNION
        SELECT StudentID, StartDate, EndDate, 'Kurs', CM.CourseID FROM
StationaryCourseMeeting
        INNER JOIN dbo.CourseModules CM on
StationaryCourseMeeting.ModuleID = CM.ModuleID
        INNER JOIN Courses ON CM.CourseID = Courses.CourseID
        INNER JOIN OrderDetails ON OrderDetails.ActivityID =
Courses.CourseID
        AND TypeOfActivity = 2
        INNER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
        WHERE StartDate > '2025-01-01'
        UNION
        SELECT StudentID, StartTime, EndTime, 'Spotkanie studyjne',
StudyMeetings.MeetingID FROM StudyMeetings
        INNER JOIN StudyMeetingPayment ON StudyMeetingPayment.MeetingID
= StudyMeetings.MeetingID
        INNER JOIN OrderDetails ON OrderDetails.DetailID =
StudyMeetingPayment.DetailID
        AND TypeOfActivity = 3
        INNER JOIN Orders ON OrderDetails.OrderID = Orders.OrderID
        WHERE StartTime > '2025-01-01'
        UNION
        SELECT Orders.StudentID, StartTime, EndTime, 'Spotkanie
studyjne', MeetingID FROM Orders
        INNER JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
        AND TypeOfActivity = 4
        INNER JOIN StudyMeetings ON StudyMeetings.MeetingID =
OrderDetails.ActivityID
        WHERE StartTime > '2025-01-01'
)
SELECT
    t2.StudentID,
    t2.StartTime AS StartTime1,
    t2.EndTime AS EndTime1,
    t2.TypeOfActivity AS Typ1,
    t2.ActivityID AS 'ID Aktywności 1',
    t3.StartTime AS StartTime2,
    t3.EndTime AS EndTime2,
    t3.TypeOfActivity AS Typ2,
    t3.ActivityID AS 'Typ Aktywności 2'
FROM
    T1 AS t2
JOIN
```

```
    T1 AS t3
    ON
        t2.StudentID = t3.StudentID
            AND t2.StartTime < t3.EndTime
            AND t2.EndTime > t3.StartTime
            AND t2.StartTime <> t3.StartTime
```

VW_allUsersStudyMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje wszystkie spotkania studyjne
- **Zawiera:** id użytkownika (StudentID), imię tego studenta (FirstName), nazwisko tego studenta (LastName), nazwa kursu (NazwaKursu), nazwa modułu kursu (NazwaModulu), data początku kursy (StartDate), data końca kursu (EndDate), status płatności (PaymentStatus)

```
CREATE VIEW VW_allUsersStudyMeetings
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT StudyName FROM Studies WHERE
Subjects.StudiesID=Studies.StudiesID) As NazwaKierunku,
SubjectName, StartTime, EndTime, StudyMeetingPayment.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3
```

VW_CurrentCoursesPassed

O.B.

OPIS

- **Funkcja**: Przechowuje informację o zdaniu modułów obecnie trwających studiów
- **Zawiera:** id kursu (CourseID), nazwa kursu (CourseName), id modułu

kursu (ModuleID), nazwa modułu (ModuleName), id użytkownika (StudentID), imię tego studenta (FirstName), nazwisko tego studenta (LastName), informację o zdaniu modułu kursu (Passed)

```
CREATE VIEW VW_CurrentCoursesPassed AS
SELECT Courses.CourseID, CourseName, CourseModules.ModuleID, ModuleName,
StudentID, FirstName, LastName, Passed FROM Courses
INNER JOIN CourseModules ON Courses.CourseID = CourseModules.CourseID
LEFT OUTER JOIN CourseModulesPassed ON CourseModules.ModuleID =
CourseModulesPassed.ModuleID
LEFT OUTER JOIN Users ON CourseModulesPassed.StudentID = Users.UserID
WHERE Courses.CourseID IN (
    SELECT id FROM VW_CoursesStartDateEndDate
    WHERE [Data rozpoczęcia] < '2025-01-01' AND [Data zakończenia] >
'2025-01-01'
)
```

VW_CourseModulesPassed

O.B.

OPIS

- **Funkcja**: Przechowuje moduły kursów zdanych przez danego studenta
- **Zawiera:** id kursu (CourseID), nazwa kursu (CourseName), id modułu kursu (ModuleID), nazwa modułu (ModuleName), id użytkownika (StudentID), imię tego studenta (FirstName), nazwisko tego studenta (LastName), informację o zdaniu modułu kursu (Passed)

```
CREATE VIEW VW_CourseModulesPassed AS
SELECT C.CourseID, CourseName, CM.ModuleID, ModuleName, StudentID,
FirstName, LastName, Passed FROM CourseModulesPassed
INNER JOIN CourseModules CM on CourseModulesPassed.ModuleID =
CM.ModuleID
INNER JOIN dbo.Courses C on CM.CourseID = C.CourseID
INNER JOIN dbo.Users U on CourseModulesPassed.StudentID = U.UserID
```

VW_Lecturers

O.B.

OPIS

- **Funkcja**: Przechowuje imiona i nazwiska wszystkich wykładowców
- **Zawiera:** imię wykładowcy (FirstName), nazwisko wykładowcy (LastName)

```
CREATE VIEW vw_Lecturers AS
SELECT FirstName, LastName
FROM Users
	INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Wykładowca')
```

VW_TranslatorWithLanguages

O.B.

OPIS

- **Funkcja**: Przechowuje imiona i nazwiska wszystkich tłumaczy oraz tłumaczony przez nich język
- **Zawiera:** imię tłumacza (FirstName), nazwisko tłumacza (LastName), tłumaczony język (LanguageName)

```
CREATE VIEW vw_TranslatorsWithLanguages AS
SELECT FirstName, LastName, LanguageName
FROM Users
	INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
	INNER JOIN TranslatedLanguage ON
TranslatedLanguage.TranslatorID=Users.UserID
	INNER JOIN Languages ON
Languages.LanguageID=TranslatedLanguage.LanguageID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Tłumacz')
```

VW_CoursesLecturers

O.B.

OPIS

- **Funkcja**: Przechowuje imiona i nazwiska wszystkich prowadzących kursów
- **Zawiera:** imię prowadzącego (FirstName), nazwisko prowadzącego

```
      (LastName)


CREATE VIEW vw_CoursesLecturers AS
SELECT FirstName, LastName
FROM Users
         INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Prowadzący kursu')
```

VW_Students

O.B.

OPIS

- **Funkcja**: Przechowuje imiona i nazwiska wszystkich studentów
- **Zawiera:** imię studenta (FirstName), nazwisko studenta (LastName)

```
CREATE VIEW vw_Students AS
SELECT FirstName, LastName
FROM Users
         INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Student')
```

---

vw_StudyCoordinator

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o koordynatorach studiów.
- **Zawiera:** Imię (FirstName) oraz nazwisko (LastName) koordynatorów.

```
CREATE VIEW vw_StudyCoordinator AS
SELECT FirstName, LastName
FROM Users
         INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Koordynator studiów')
```

---

vw_WebinarTeachers

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o prowadzących webinary.
- **Zawiera:** Imię (FirstName) oraz nazwisko (LastName) prowadzących.

```
CREATE VIEW vw_WebinarTeachers AS
SELECT FirstName, LastName
FROM Users
         INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Prowadzący webinaru')
```

---

vw_CourseCoordinators

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o koordynatorach kursów.
- **Zawiera:** Imię (FirstName) oraz nazwisko (LastName) koordynatorów.

```
CREATE VIEW vw_CourseCoordinators AS
SELECT FirstName, LastName
FROM Users
         INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Koordynator kursu')
```

---

vw_InternshipCoordinators

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o koordynatorach praktyk.
- **Zawiera:** Imię (FirstName) oraz nazwisko (LastName) koordynatorów.

```
CREATE VIEW vw_InternshipCoordinators AS
SELECT FirstName, LastName
FROM Users
         INNER JOIN UsersRoles ON UsersRoles.UserID=Users.UserID
WHERE UsersRoles.RoleID = (SELECT RoleID FROM Roles WHERE
RoleName='Koordynator praktyk')
```

---

vw_NumberOfHoursOfWOrkForAllEmployees

J.K.

OPIS:

- Funkcja: Przechowuje informacje o liczbie godzin przepracowanych przez każdego pracownika
- Zawiera: id pracownika (EmployeeID) oraz liczbę godzin przez niego przepracowanych przez cały okres jego zatrudnienia (liczbagodzinpracy)

```
CREATE VIEW vw_NumberOfHoursOfWOrkForAllEmployees as
with t1 as (
select EmployeeID, (ISNULL(DATEDIFF(minute,sm.EndTime , sm.StartTime), 0)+
ISNULL(DATEDIFF(minute,sm1.EndTime, sm1.StartTime),
0)+ISNULL(DATEDIFF(minute,w.EndDate,w.StartDate), 0)+
ISNULL(DATEDIFF(minute,w1.EndDate, w1.StartDate),
0)+ISNULL(DATEDIFF(minute,ocm.EndDate, ocm.StartDate),
0)+ISNULL(DATEDIFF(minute,ocm1.EndDate, ocm1.StartDate),
0)+ISNULL(DATEDIFF(minute,scm.EndDate,scm.StartDate),
0)+ISNULL(DATEDIFF(minute,scm1.EndDate,scm1.StartDate), 0)) * (-1) as
liczbaminut from Employees
left outer join StudyMeetings as sm on sm.LecturerID =
Employees.EmployeeID
left outer join StudyMeetings as sm1 on sm1.TranslatorID =
Employees.EmployeeID
left outer join Webinars as w on w.TeacherID = Employees.EmployeeID
left outer join Webinars as w1 on w1.TranslatorID = Employees.EmployeeID
left outer join CourseModules as cm on cm.LecturerID =
Employees.EmployeeID
left outer join CourseModules as cm1 on cm1.TranslatorID =
Employees.EmployeeID
left outer join OnlineCourseMeeting as ocm on ocm.ModuleID = cm.ModuleID
```

```
left outer join OnlineCourseMeeting as ocm1 on ocm1.ModuleID =
cm1.ModuleID
left outer join StationaryCourseMeeting as scm on scm.ModuleID =
cm.ModuleID
left outer join StationaryCourseMeeting as scm1 on scm1.ModuleID =
cm1.ModuleID
UNION
select internshipCoordinatorID, 3000 from internship
)

select employeeid, round(SUM(liczbaminut)/60,2) as liczbagodzinpracy from
t1
group by EmployeeID
```

VW_allUsersWebinars

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich webinarach, na które są zapisani poszczególni użytkownicy.
- **Zawiera:** ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwa webinaru (WebinarName), początek webinaru (StartDate), koniec webinaru (EndDate), status płatności użytkownika za webinar (PaymentStatus)

```
CREATE VIEW VW_allUsersWebinars AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
Webinars.WebinarName, Webinars.StartDate, Webinars.EndDate,
OrderDetails.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1
```

VW_allUsersMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich spotkaniach, na które są zapisani poszczególni użytkownicy.
- **Zawiera:** ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), typ aktywności w ramach której jest spotkanie (ActivityType), nazwę aktywności (ActivityName), początek spotkania (StartDate), koniec spotkania (EndTime), status płatności użytkownika za spotkanie (PaymentStatus)

```
CREATE VIEW VW_allUsersMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As ActivityName,
StartTime, EndTime, StudyMeetingPayment.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    StationaryCourseMeeting.StartDate as StartTime,
    StationaryCourseMeeting.EndDate as EndTime,
    OrderDetails.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON
ActivitiesTypes.ActivityTypeID=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
```

```
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    OnlineCourseMeeting.StartDate as StartTime,
    OnlineCourseMeeting.EndDate as EndTime,
    OrderDetails.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON OrderDetails.TypeOfActivity =
ActivitiesTypes.ActivityTypeID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2

UNION

SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
Webinars.WebinarName as ActivityName, Webinars.StartDate as StartTime,
Webinars.EndDate as EndTime, OrderDetails.PaymentStatus
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1
```

VW_allUsersPastStudyMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przeszłych spotkaniach studyjnych, na które są zapisani poszczególni użytkownicy.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę studiów (StudyName), nazwę przedmiotu (SubjectName), początek spotkania (StartTime), koniec spotkania (EndTime)

```
CREATE VIEW VW_allUsersPastStudyMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As NazwaKierunku,
SubjectName, StartTime, EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3 AND StudyMeetings.EndTime<GETDATE()
```

VW_allUsersPastCourseMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przeszłych kursach, poszczególnych użytkowników.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę kursów (CourseName), nazwę modułu (ModuleName), początek spotkania (StartTime), koniec spotkania (EndTime)

```
CREATE VIEW VW_allUsersPastCourseMeetings AS
SELECT
    Users.UserID,
    Users.FirstName,
```

```
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    StationaryCourseMeeting.StartDate,
    StationaryCourseMeeting.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND EndDate<GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    OnlineCourseMeeting.StartDate,
    OnlineCourseMeeting.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND EndDate<GETDATE()
```

VW_allPastUsersWebinars

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przeszłych webinarach poszczególnych użytkowników.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę webinaru (WebinarName), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allPastUsersWebinars AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
Webinars.WebinarName, Webinars.StartDate, Webinars.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND EndDate<GETDATE()
```

VW_allUsersPastMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przeszłych spotkaniach, na które byli zapisani użytkownicy.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę typu aktywności (ActivityType), nazwę aktywności (ActivityName), początek (StartTime), koniec (EndTime)

```
CREATE VIEW VW_allUsersPastMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As ActivityName,
StartTime, EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3 AND EndTime < GETDATE()

UNION

SELECT
    Users.UserID,
```

```
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    StationaryCourseMeeting.StartDate as StartTime,
    StationaryCourseMeeting.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON
ActivitiesTypes.ActivityTypeID=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND EndDate < GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    OnlineCourseMeeting.StartDate as StartTime,
    OnlineCourseMeeting.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON OrderDetails.TypeOfActivity =
ActivitiesTypes.ActivityTypeID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND EndDate < GETDATE()

UNION

SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
```

```
ActivityType,
Webinars.WebinarName as ActivityName, Webinars.StartDate as StartTime,
Webinars.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND EndDate < GETDATE()
```

VW_allUsersFutureStudyMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przyszłych spotkaniach studyjnych, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę kierunku (StudyName), nazwę przedmiotu (SubjectName), początek (StartTime), koniec (EndTime)

```
CREATE VIEW VW_allUsersFutureStudyMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As NazwaKierunku,
SubjectName, StartTime, EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3 AND StudyMeetings.StartTime>GETDATE()
```

VW_allUsersFutureCourseMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przyszłych spotkaniach

w ramach kursu, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę kursu (NazwaKursu), nazwę modułu kursu (NazwaModulu), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allUsersFutureCourseMeetings AS
SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    StationaryCourseMeeting.StartDate,
    StationaryCourseMeeting.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate>GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    OnlineCourseMeeting.StartDate,
    OnlineCourseMeeting.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate>GETDATE()
```

VW_allUsersFutureCourseMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przyszłychwebinarach, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę webinaru (WebinarName), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allUsersFutureWebinars AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
Webinars.WebinarName, Webinars.StartDate, Webinars.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND StartDate>GETDATE()
```

VW_allUsersFutureMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przyszłych spotkaniach, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę typu aktywności (ActivityType), nazwę aktywności (ActivityName), początek (StartTime), koniec (EndTime)

```
CREATE VIEW VW_allUsersFutureMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As ActivityName,
StartTime, EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
```

```
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3 AND StartTime > GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    StationaryCourseMeeting.StartDate as StartTime,
    StationaryCourseMeeting.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON
ActivitiesTypes.ActivityTypeID=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate > GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    OnlineCourseMeeting.StartDate as StartTime,
    OnlineCourseMeeting.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON OrderDetails.TypeOfActivity =
ActivitiesTypes.ActivityTypeID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
```

```
WHERE OrderDetails.TypeOfActivity=2 AND StartDate > GETDATE()

UNION

SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
Webinars.WebinarName as ActivityName, Webinars.StartDate as StartTime,
Webinars.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND StartDate > GETDATE()
```

VW_allUsersCurrentStudyMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich obecnie trwających spotkaniach studyjnych, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę kierunku (NazwaKierunku), nazwę przedmiotu (SubjectName), początek (StartTime), koniec (EndTime)

```
CREATE VIEW VW_allUsersCurrentStudyMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As NazwaKierunku,
SubjectName, StartTime, EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
WHERE OrderDetails.TypeOfActivity=3 AND StudyMeetings.StartTime<GETDATE()
AND StudyMeetings.StartTime>GETDATE()
```

VW_allUsersCurrentCourseMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich aktualnych spotkaniach w ramach kursów, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę kursu (NazwaKursu), nazwę modułu kursu (NazwaModulu), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allUsersCurrentCourseMeetings AS
SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    StationaryCourseMeeting.StartDate,
    StationaryCourseMeeting.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate<GETDATE() AND
EndDate>GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    Courses.CourseName AS NazwaKursu,
    CourseModules.ModuleName AS NazwaModulu,
    OnlineCourseMeeting.StartDate,
    OnlineCourseMeeting.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
```

```
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate<GETDATE() AND
EndDate>GETDATE()
```

VW_allUsersCurrentWebinars

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich aktualnych webinarach, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), nazwę webinaru (WebinarName), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allUsersCurrentWebinars AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
Webinars.WebinarName, Webinars.StartDate, Webinars.EndDate
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND StartDate<GETDATE() AND EndDate>GETDATE()
```

VW_allUsersCurrentMeetings

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich aktualnie trwających spotkaniach, na które jest zapisany użytkownik.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), typ aktywności (ActivityType), nazwę aktywności (ActivityName), początek (StartTime), koniec (EndTime)

```
CREATE VIEW VW_allUsersCurrentMeetings AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
```

```
ActivityType,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As ActivityName,
StartTime, EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
LEFT JOIN StationaryMeetings ON
StationaryMeetings.MeetingID=StudyMeetings.MeetingID
WHERE OrderDetails.TypeOfActivity=3 AND StartTime<GETDATE() AND
EndTime>GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    StationaryCourseMeeting.StartDate as StartTime,
    StationaryCourseMeeting.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON
ActivitiesTypes.ActivityTypeID=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate<GETDATE() AND
EndDate>GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
```

```
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    OnlineCourseMeeting.StartDate as StartTime,
    OnlineCourseMeeting.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON OrderDetails.TypeOfActivity =
ActivitiesTypes.ActivityTypeID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN OnlineCourseMeeting ON OnlineCourseMeeting.ModuleID =
CourseModules.ModuleID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate < GETDATE() AND EndDate
> GETDATE()

UNION

SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
Webinars.WebinarName as ActivityName, Webinars.StartDate as StartTime,
Webinars.EndDate as EndTime
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON Orders.OrderID=OrderDetails.OrderID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND StartDate < GETDATE() AND EndDate > GETDATE()
```

VW_allUsersStationaryMeetingsWithRoomAndAddresses

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich aktualnie trwających stacjonarnych spotkaniach, na które jest zapisany użytkownik wraz z salą i adresem.

Zawiera: ID użytkownika (UserID), imię (FirstName), nazwisko (LastName), typ aktywności (ActivityType), nazwę aktywności (ActivityName), początek (StartTime), koniec (EndTime), nazwę sali (RoomName), adress (Street,

```
PostalCode, CityName)

CREATE VIEW VW_allUsersStationaryMeetingsWithRoomAndAddresses AS
SELECT Users.UserID, Users.FirstName, Users.LastName,
(SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
(SELECT StudyName FROM Studies WHERE Subjects.StudiesID=Studies.StudiesID)
As ActivityName,
StartTime, EndTime, Rooms.RoomName, Rooms.Street, PostalCode, CityName
FROM Users
INNER JOIN Orders ON Orders.StudentID=Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN StudyMeetingPayment ON
StudyMeetingPayment.DetailID=OrderDetails.DetailID
INNER JOIN StudyMeetings ON
StudyMeetings.MeetingID=StudyMeetingPayment.MeetingID
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
LEFT JOIN StationaryMeetings ON
StationaryMeetings.MeetingID=StudyMeetings.MeetingID
LEFT JOIN Rooms ON Rooms.RoomID=StationaryMeetings.RoomID
LEFT JOIN Cities ON Cities.CityID=Rooms.CityID
WHERE OrderDetails.TypeOfActivity=3 AND StartTime<GETDATE() AND
EndTime>GETDATE()

UNION

SELECT
    Users.UserID,
    Users.FirstName,
    Users.LastName,
    (SELECT TypeName FROM ActivitiesTypes WHERE
OrderDetails.TypeOfActivity=ActivitiesTypes.ActivityTypeID) as
ActivityType,
    Courses.CourseName AS ActivityName,
    StationaryCourseMeeting.StartDate as StartTime,
    StationaryCourseMeeting.EndDate as EndTime,
    Rooms.RoomName, Rooms.Street, PostalCode, CityName
FROM Users
INNER JOIN Orders ON Orders.StudentID = Users.UserID
INNER JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
INNER JOIN ActivitiesTypes ON
ActivitiesTypes.ActivityTypeID=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID = OrderDetails.ActivityID
INNER JOIN CourseModules ON CourseModules.CourseID = Courses.CourseID
INNER JOIN StationaryCourseMeeting ON StationaryCourseMeeting.ModuleID =
CourseModules.ModuleID
```

```
INNER JOIN Rooms ON Rooms.RoomID=StationaryCourseMeeting.RoomID
INNER JOIN Cities ON Cities.CityID=Rooms.CityID
WHERE OrderDetails.TypeOfActivity=2 AND StartDate<GETDATE() AND
EndDate>GETDATE()
```

VW_StudiesStartDateEndDate

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o początkach i końcach studiów.

Zawiera: ID studiów (StudiesID), początek (startDate), koniec (endDate)

```
CREATE VIEW VW_StudiesStartDateEndDate AS
With t1 as
(SELECT S2.StudiesID, CONCAT(DATENAME(MONTH,
MIN(StudyMeetings.StartTime)),' ' + DATENAME(YEAR,
MIN(StudyMeetings.StartTime))) as startDate
        FROM StudyMeetings
                INNER JOIN Subjects ON
Subjects.SubjectID=StudyMeetings.SubjectID
                INNER JOIN Studies S2 ON S2.StudiesID=Subjects.StudiesID
                GROUP BY S2.StudiesID)
SELECT S2.StudiesID, t1.startDate, CONCAT(DATENAME(MONTH,
MAX(StudyMeetings.EndTime)),' ' + DATENAME(YEAR,
MAX(StudyMeetings.EndTime))) as endDate
FROM StudyMeetings
INNER JOIN Subjects ON Subjects.SubjectID=StudyMeetings.SubjectID
INNER JOIN Studies S2 ON S2.StudiesID=Subjects.StudiesID
INNER JOIN t1 ON t1.studiesID=S2.StudiesID
GROUP BY S2.StudiesID, t1.startDate
```

VW_allOrderedActivities

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich aktywnościach, na które użytkownicy złożyli zamówienia

Zawiera: ID aktywności (ActivityID), typ aktywności (ActivityType), nazwę aktywności (ActivityName), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allOrderedActivities AS
SELECT ActivityID,
(SELECT typename FROM ActivitiesTypes WHERE
ActivitiesTypes.ActivityTypeID=OrderDetails.TypeOfActivity) as
ActivityType,
Studies.StudyName as ActivityName,
vw.StartDate, vw.EndDate
FROM Orders
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN VW_StudiesStartDateEndDate vw ON
vw.studiesid=OrderDetails.ActivityID
INNER JOIN Studies ON Studies.StudiesID=OrderDetails.ActivityID
WHERE TypeOfActivity=3
UNION
SELECT ActivityID,
(SELECT typename FROM ActivitiesTypes WHERE
ActivitiesTypes.ActivityTypeID=OrderDetails.TypeOfActivity) as
ActivityType,
Courses.CourseName as ActivityName,
DATENAME(DAY, vw.[Data rozpoczęcia])+' '+DATENAME(MONTH, vw.[Data
rozpoczęcia])+' '+ DATENAME(YEAR, vw.[Data rozpoczęcia]) as StartDate,
DATENAME(DAY, vw.[Data rozpoczęcia])+' '+DATENAME(MONTH,vw.[Data
zakończenia]) + ' ' +DATENAME(YEAR,vw.[Data zakończenia]) as EndDate
FROM Orders
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN VW_CoursesStartDateEndDate vw ON vw.id=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID=OrderDetails.ActivityID
WHERE TypeOfActivity=2
UNION
SELECT ActivityID,
(SELECT typename FROM ActivitiesTypes WHERE
ActivitiesTypes.ActivityTypeID=OrderDetails.TypeOfActivity) as
ActivityType,
Webinars.WebinarName as ActivityName,
DATENAME(DAY,StartDate)+' '+DATENAME(MONTH,StartDate)+'
'+DATENAME(YEAR,StartDate)+'
'+DATENAME(HOUR,StartDate)+':'+DATENAME(MINUTE,StartDate),
DATENAME(DAY,EndDate)+' '+DATENAME(MONTH,EndDate)+'
'+DATENAME(YEAR,EndDate)+'
'+DATENAME(HOUR,EndDate)+':'+DATENAME(MINUTE,EndDate)
FROM Orders
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN VW_CoursesStartDateEndDate vw ON vw.id=OrderDetails.ActivityID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1
```

VW_allFutureOrderedActivities

O.B.

OPIS

- **Funkcja**: Przechowuje informacje o wszystkich przyszłych aktywnościach, na które już złożono jakieś zamówienie.

Zawiera: ID aktywności (ActivityID), typ aktywności (ActivityType), nazwę aktywności (ActivityName), początek (StartDate), koniec (EndDate)

```
CREATE VIEW VW_allFutureOrderedActivities AS
SELECT ActivityID,
(SELECT typename FROM ActivitiesTypes WHERE
ActivitiesTypes.ActivityTypeID=OrderDetails.TypeOfActivity) as
ActivityType,
Studies.StudyName as ActivityName,
vw.StartDate, vw.EndDate
FROM Orders
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN VW_StudiesStartDateEndDate vw ON
vw.studiesid=OrderDetails.ActivityID
INNER JOIN Studies ON Studies.StudiesID=OrderDetails.ActivityID
WHERE TypeOfActivity=3 AND StartDate>GETDATE()
UNION
SELECT ActivityID,
(SELECT typename FROM ActivitiesTypes WHERE
ActivitiesTypes.ActivityTypeID=OrderDetails.TypeOfActivity) as
ActivityType,
Courses.CourseName as ActivityName,
DATENAME(DAY, vw.[Data rozpoczęcia])+' '+DATENAME(MONTH, vw.[Data
rozpoczęcia])+' '+ DATENAME(YEAR, vw.[Data rozpoczęcia]) as StartDate,
DATENAME(DAY, vw.[Data rozpoczęcia])+' '+DATENAME(MONTH,vw.[Data
zakończenia]) + ' ' +DATENAME(YEAR,vw.[Data zakończenia]) as EndDate
FROM Orders
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN VW_CoursesStartDateEndDate vw ON vw.id=OrderDetails.ActivityID
INNER JOIN Courses ON Courses.CourseID=OrderDetails.ActivityID
WHERE TypeOfActivity=2 AND
DATENAME(DAY, vw.[Data rozpoczęcia])+' '+DATENAME(MONTH, vw.[Data
rozpoczęcia])+' '+ DATENAME(YEAR, vw.[Data rozpoczęcia]) >GETDATE()
UNION
SELECT ActivityID,
(SELECT typename FROM ActivitiesTypes WHERE
ActivitiesTypes.ActivityTypeID=OrderDetails.TypeOfActivity) as
```

```
ActivityType,
Webinars.WebinarName as ActivityName,
DATENAME(DAY,StartDate)+' '+DATENAME(MONTH,StartDate)+'
'+DATENAME(YEAR,StartDate)+'
'+DATENAME(HOUR,StartDate)+':'+DATENAME(MINUTE,StartDate),
DATENAME(DAY,EndDate)+' '+DATENAME(MONTH,EndDate)+'
'+DATENAME(YEAR,EndDate)+'
'+DATENAME(HOUR,EndDate)+':'+DATENAME(MINUTE,EndDate)
FROM Orders
INNER JOIN OrderDetails ON OrderDetails.OrderID=Orders.OrderID
INNER JOIN VW_CoursesStartDateEndDate vw ON vw.id=OrderDetails.ActivityID
INNER JOIN Webinars ON Webinars.WebinarID=OrderDetails.ActivityID
WHERE TypeOfActivity=1 AND
DATENAME(DAY,StartDate)+' '+DATENAME(MONTH,StartDate)+'
'+DATENAME(YEAR,StartDate)+'
'+DATENAME(HOUR,StartDate)+':'+DATENAME(MINUTE,StartDate) > GETDATE()
```

VW_RemaingSeats

K.B.

OPIS

- **Funkcja**: Przechowuje informacje o dostępnych miejscach na kursy, studia i pojedyncze stacjonarne spotkania studyjne. Null wskazuje na brak limitu (wydarzenie w pełni online)

Zawiera: Nazwę typu aktywności (activityType), ID aktywności (id), nazwę aktywności (ActivityName), liczbę dostępnych miejsc (Available), początek (Start)

```
CREATE VIEW VW_RemainingSeats AS
SELECT 'kurs' as 'activityType', c.coursename as activityName,
c.courseid as id, c.StudentLimit-(SELECT ISnull(Count(*),0) FROM
OrderDetails od where od.ActivityID=c.courseid and od.TypeOfActivity=2)
as available, FORMAT(starts.[Data rozpoczęcia], 'yyyy-MM-dd HH:mm') as
start
FROM courses c
inner join VW_CoursesStartDateEndDate starts on starts.id = c.CourseID
UNION
SELECT 'studia' as 'activityType', s.StudyName as activityName,
s.StudiesID as id, s.StudentLimit - (SELECT ISNULL(COUNT(*), 0) FROM
OrderDetails OD WHERE OD.ActivityID = s.StudiesID and OD.TypeOfActivity
= 3) as available, starts.startDate as start
FROM Studies s
inner join VW_StudiesStartDateEndDate starts on
```

```
starts.StudiesID=s.StudiesID
UNION
SELECT 'spotkanie studyjne' as 'activityType', sub.subjectname as name,
sm.MeetingID as id, isnull(sm.StudentLimit,0) - (SELECT ISNULL(COUNT(*),
0) as available
             FROM StudyMeetingPayment smp
                   where smp.MeetingID = sm.MeetingID)-
                   (select count(*) from orderdetails od where
od.ActivityID=sm.meetingid and od.TypeOfActivity=4)  as available,
FORMAT(studmeet.StartTime, 'yyyy-MM-dd HH:mm') as start
FROM StationaryMeetings sm
inner join StudyMeetings studmeet on studmeet.MeetingID=sm.MeetingID
inner join Subjects sub on sub.Subjectid=studmeet.SubjectID
```

VW_FutureEventsWithDetails

O.B.

Funkcja:
Widok przechowujący szczegółowe informacje o przyszłych wydarzeniach, które obejmują kursy, webinary oraz studia, które mają rozpoczęcie po bieżącej dacie.

Zawiera:

- **EventType** – Typ wydarzenia (np. Kurs, Webinar, Study)
- **EventName** – Nazwa wydarzenia
- **StartDate** – Data rozpoczęcia wydarzenia
- **EndDate** – Data zakończenia wydarzenia

```
CREATE VIEW VW_FutureEventsWithDetails AS
-- Kursy
SELECT
      'Course' AS EventType,
      c.CourseName AS EventName,
      vce.[Data rozpoczęcia] AS StartDate,
      vce.[Data zakończenia] AS EndDate
FROM
      Courses c
      JOIN
      VW_CoursesStartDateEndDate vce ON c.CourseID = vce.ID
WHERE
```

```
        vce.[Data rozpoczęcia] > GETDATE()

UNION ALL

-- Webinary
SELECT
        'Webinar' AS EventType,
        w.WebinarName AS EventName,
        w.StartDate AS StartDate,
        w.EndDate AS EndDate
FROM
        Webinars w
WHERE
        w.StartDate > GETDATE()

UNION ALL

-- Studia
SELECT
        'Study' AS EventType,
        s.StudyName AS EventName,
        vsde.StartDate AS StartDate,
        vsde.EndDate AS EndDate
FROM
        Studies s
        JOIN
        VW_StudiesStartDateEndDate vsde ON s.StudiesID = vsde.StudiesID
WHERE
        vsde.StartDate > GETDATE();
```

PROCEDURY:

AddNewStudy

O.B.

Cel: Dodanie nowego kierunku studiów do tabeli `studies`.

Parametry:

- `@StudyName NVARCHAR(255)`: Nazwa kierunku studiów, który ma zostać dodany.
- `@StudiesCoordinatorID INT`: ID koordynatora odpowiedzialnego za kierunek studiów. Wartość musi istnieć w tabeli `employees`.

- `@StudyPrice DECIMAL(10, 2)`: Cena za kierunek studiów. Nie może być wartością ujemną.
- `@NumberOfTerms INT`: Liczba semestrów kierunku studiów. Musi być większa od zera.

Walidacje:

1. Sprawdzenie, czy podany `@StudiesCoordinatorID` istnieje w tabeli `employees`.
2. Sprawdzenie, czy liczba semestrów (`@NumberOfTerms`) jest większa od zera.
3. Sprawdzenie, czy cena studiów (`@StudyPrice`) nie jest ujemna.

```
CREATE PROCEDURE Addnewstudy @StudyName            NVARCHAR(255),
                             @StudyDescription     NVARCHAR(255),
                             @StudiesCoordinatorID INT,
                             @StudyPrice           DECIMAL(10, 2),
                             @NumberOfTerms        INT,
                             @StudentLimit         INT
AS
  BEGIN
      BEGIN try
          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @StudiesCoordinatorID)
            BEGIN
                RAISERROR('Koordynator o podanym ID nie istnieje.',16,1);

                RETURN;
            END

          IF @NumberOfTerms <= 0
            BEGIN
                RAISERROR('Liczba semestrów musi być większa od
zera.',16,1);

                RETURN;
            END

          IF @StudyPrice < 0
            BEGIN
                RAISERROR('Cena studiów nie może być ujemna.',16,1);

                RETURN;
            END

          INSERT INTO studies
```

```
                    (studyname,
                     studydescription,
                     studiescoordinatorid,
                     studyprice,
                     numberofterms,
                     studentlimit)
        VALUES      (@StudyName,
                     @StudyDescription,
                     @StudiesCoordinatorID,
                     @StudyPrice,
                     @NumberOfTerms,
                     @StudentLimit);

        PRINT 'Studia zostały pomyślnie dodane.';
    END try

    BEGIN catch
        PRINT 'Wystąpił błąd podczas dodawania studiów.';

        PRINT Error_message();
    END catch
END;
```

AddNewCourse

O.B.

Cel:
Procedura dodaje nowy kurs do tabeli `courses` w bazie danych. Sprawdza poprawność wprowadzonych danych i obsługuje potencjalne błędy.

Parametry:

1. **@CourseName** (`NVARCHAR(255)`)
   Nazwa kursu.
2. **@CourseCoordinatorID** (`INT`)
   ID koordynatora kursu, musi istnieć w tabeli `employees`.
3. **@CoursePrice** (`DECIMAL(10, 2)`)
   Cena kursu, nie może być ujemna.
4. **@StudentLimit** (`INT`)
   Limit studentów, musi być większy od zera lub wartością `NULL`.

Walidacje:

- Sprawdza, czy podany `@CourseCoordinatorID` istnieje w tabeli `employees`.
- Weryfikuje, czy `@StudentLimit` jest większy od zera lub równy `NULL`.
- Sprawdza, czy `@CoursePrice` nie jest ujemna.

```
CREATE PROCEDURE Addnewcourse @CourseName          NVARCHAR(255),
                              @CourseCoordinatorID INT,
                              @CoursePrice         DECIMAL(10, 2),
                              @StudentLimit        INT
AS
  BEGIN
      BEGIN try
          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @CourseCoordinatorID)
            BEGIN
                RAISERROR('Koordynator kursu o podanym ID nie
istnieje.',16,1);

                RETURN;
            END

          IF @StudentLimit <= 0
             AND @StudentLimit IS NOT NULL
            BEGIN
                RAISERROR(
                'Limit studentów musi być większy od zera lub NULL.',
                16,1
                );

                RETURN;
            END

          IF @CoursePrice < 0
            BEGIN
                RAISERROR('Cena kursu nie może być ujemna.',16,1);

                RETURN;
            END

          INSERT INTO courses
                      (coursename,
                       coursecoordinatorid,
                       courseprice,
                       studentlimit)
          VALUES      (@CourseName,
```

```
                    @CourseCoordinatorID,
                    @CoursePrice,
                    @StudentLimit);

        PRINT 'Kurs został pomyślnie dodany.';
    END try

    BEGIN catch
        PRINT 'Wystąpił błąd podczas dodawania kursu.';

        PRINT Error_message();
    END catch
END;
```

AddNewWebinar

`J.K.`

Cel:
Procedura dodaje nowy webinar do tabeli `webinars` w bazie danych. Weryfikuje poprawność wprowadzonych danych, w tym zakres dat i powiązania z innymi tabelami.

Parametry:

1. **@WebinarName** (`NVARCHAR(255)`)
   Nazwa webinaru.
2. **@StartDate** (`DATETIME`)
   Data rozpoczęcia webinaru.
3. **@EndDate** (`DATETIME`)
   Data zakończenia webinaru, musi być późniejsza niż data rozpoczęcia.
4. **@TeacherID** (`INT`)
   ID nauczyciela prowadzącego webinar, musi istnieć w tabeli `employees`.
5. **@TranslatorID** (`INT`, opcjonalne)
   ID tłumacza, jeśli istnieje, musi znajdować się w tabeli `employees`.
6. **@LanguageID** (`INT`)
   ID języka, musi istnieć w tabeli `languages` i być dostępny w tabeli `translatedlanguage`.
7. **@Price** (`DECIMAL(10, 2)`, domyślnie `0.0`)
   Cena webinaru, nie może być ujemna.

Walidacje:

- Sprawdza, czy `@StartDate` jest wcześniejsza niż `@EndDate`.
- Weryfikuje istnienie `@TeacherID` w tabeli `employees`.
- Sprawdza, czy `@TranslatorID` (jeśli podano) istnieje w tabeli `employees`.

- Weryfikuje, czy `@LanguageID` istnieje w tabeli `languages` i jest dostępny w tabeli `translatedlanguage`.
- Sprawdza, czy `@Price` nie jest ujemna

```
CREATE PROCEDURE Addnewwebinar @WebinarName  NVARCHAR(255),

                               @WebinarDescription  NVARCHAR(255),
                               @StartDate    DATETIME,
                               @EndDate      DATETIME,
                               @TeacherID    INT,
                               @TranslatorID INT = NULL,
                               @LanguageID   INT,
                               @Price        DECIMAL(10, 2) = 0.0
AS
  BEGIN
      BEGIN try
          IF @StartDate >= @EndDate
            BEGIN
                RAISERROR(
      'Data rozpoczęcia musi być wcześniejsza niż data zakończenia.',16
      ,1);

      RETURN;
            END

          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @TeacherID)
            BEGIN
                RAISERROR('Nauczyciel o podanym ID nie istnieje.',16,1);

                RETURN;
            END

          IF (@LanguageID <> 1) AND (@LanguageID NOT IN (SELECT languageid
                                FROM   translatedlanguage))
            BEGIN
                THROW 67007, 'Language not available.', 1;
            END

          IF @TranslatorID IS NOT NULL
             AND NOT EXISTS (SELECT 1
                             FROM   employees
                             WHERE  employeeid = @TranslatorID)
            BEGIN
```

```
            RAISERROR('Tłumacz o podanym ID nie istnieje.',16,1);

            RETURN;
        END

    IF NOT EXISTS (SELECT 1
                   FROM   languages
                   WHERE  languageid = @LanguageID)
        BEGIN
            RAISERROR('Język o podanym ID nie istnieje.',16,1);

            RETURN;
        END

    IF @Price < 0
        BEGIN
            RAISERROR('Cena webinaru nie może być ujemna.',16,1);

            RETURN;
        END

    INSERT INTO webinars
                (webinarname,

                 webinardescription
                 startdate,
                 enddate,
                 teacherid,
                 translatorid,
                 languageid,
                 price)
    VALUES      (@WebinarName,

                 @WebinarDescription
                 @StartDate,
                 @EndDate,
                 @TeacherID,
                 @TranslatorID,
                 @LanguageID,
                 @Price);

    PRINT 'Webinar został pomyślnie dodany.';
END try

BEGIN catch
    PRINT 'Wystąpił błąd podczas dodawania webinaru.';
```

```
        PRINT Error_message();
    END catch
END;
```

AddNewSubject

O.B.

Cel:
Procedura dodaje nowy przedmiot do tabeli `subjects` w bazie danych. Weryfikuje poprawność danych związanych z przedmiotem, jego godzinami oraz powiązaniem ze studiami.

Parametry:

1. **@SubjectName** (`NVARCHAR(255)`)
   Nazwa przedmiotu.
2. **@NumberOfHoursInTerm** (`INT`)
   Liczba godzin w semestrze, musi być większa od zera.
3. **@TeacherID** (`INT`)
   ID nauczyciela prowadzącego, musi istnieć w tabeli `employees`.
4. **@StudiesID** (`INT`)
   ID studiów, musi istnieć w tabeli `studies`.

Walidacje:

- Sprawdza, czy `@NumberOfHoursInTerm` jest większe od zera i mieści się w liczbie semestrów w tabeli `studies`.
- Weryfikuje istnienie `@TeacherID` w tabeli `employees`.
- Sprawdza, czy `@StudiesID` istnieje w tabeli `studies`.

```
CREATE PROCEDURE Addnewsubject @SubjectName         NVARCHAR(255),
                               @SubjectDescription  NVARCHAR(255),
                               @TermNumber          INT,
                               @NumberOfHoursInTerm INT,
                               @TeacherID           INT,
                               @StudiesID           INT
AS
  BEGIN
      BEGIN try
          IF @NumberOfHoursInTerm <= 0
             AND @NumberOfHoursInTerm > (SELECT numberofterms
                                         FROM   studies
                                         WHERE  studiesid = @StudiesID)
            BEGIN
```

```
            RAISERROR(
            'Liczba godzin w semestrze musi być większa od zera.',
            16,1
            );

            RETURN;
        END

    IF NOT EXISTS (SELECT 1
                   FROM   employees
                   WHERE  employeeid = @TeacherID)
        BEGIN
            RAISERROR('Nauczyciel o podanym ID nie istnieje.',16,1);

            RETURN;
        END

    IF NOT EXISTS (SELECT 1
                   FROM   studies
                   WHERE  studiesid = @StudiesID)
        BEGIN
            RAISERROR('Studia o podanym ID nie istnieją.',16,1);

            RETURN;
        END

    INSERT INTO subjects
                (subjectname,
                 subjectdescription,
                 numberofhoursinterm,
                 teacherid,
                 studiesid,
                 term)
    VALUES      (@SubjectName,
                 @SubjectDescription,
                 @NumberOfHoursInTerm,
                 @TeacherID,
                 @StudiesID,
                 @TermNumber);

    PRINT 'Przedmiot został pomyślnie dodany.';
END try

BEGIN catch
    PRINT 'Wystąpił błąd podczas dodawania przedmiotu.';

    PRINT Error_message();
```

```
    END catch
END;
```

AddCourseModule

O.B.

Cel:
Procedura dodaje nowy moduł do kursu w tabeli `coursemodules` w bazie danych. Sprawdza poprawność wprowadzonych danych i obsługuje potencjalne błędy.

Parametry:

1. **@CourseID** (`INT`)
   ID kursu, do którego ma zostać dodany moduł.
2. **@ModuleName** (`NVARCHAR(255)`)
   Nazwa modułu.
3. **@ModuleType** (`INT`)
   Typ modułu (odwołanie do tabeli `formofactivity` w celu weryfikacji typu).
4. **@LecturerID** (`INT`)
   ID wykładowcy odpowiedzialnego za moduł.
5. **@TranslatorID** (`INT`, opcjonalne)
   ID tłumacza, jeżeli istnieje.
6. **@LanguageID** (`INT`)
   ID języka, który jest dostępny w tabeli `translatedlanguage`.

Walidacje:

- Sprawdza, czy `@LanguageID` istnieje w tabeli `translatedlanguage`.
- Weryfikuje, czy `@ModuleType` jest poprawny i istnieje w tabeli `formofactivity`.

```
CREATE PROCEDURE Addcoursemodule @CourseID     INT,
                                 @ModuleName   NVARCHAR(255),
                                 @ModuleType   INT,
                                 @LecturerID   INT,
                                 @TranslatorID INT = NULL,
                                 @LanguageID   INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF (@LanguageID <> 1) AND (@LanguageID NOT IN (SELECT languageid
                                 FROM   translatedlanguage))
            BEGIN
                THROW 67007, 'Language not available.', 1;
```

```
            END;

        IF NOT EXISTS (SELECT 1
                       FROM   formofactivity
                       WHERE  activitytypeid = @ModuleType)
          BEGIN
              THROW 50001, 'Nieprawidłowy typ modułu.', 1;
          END;

        INSERT INTO coursemodules
                    (courseid,
                     modulename,
                     moduletype,
                     lecturerid,
                     translatorid,
                     languageid)
        VALUES      (@CourseID,
                     @ModuleName,
                     @ModuleType,
                     @LecturerID,
                     @TranslatorID,
                     @LanguageID);

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch
END;
```

AddCourseMeeting

O.B.

Cel:
Procedura dodaje spotkanie dla modułu kursu do odpowiedniej tabeli (`stationarycoursemeeting` lub `onlinecoursemeeting`). Sprawdza poprawność danych oraz typ spotkania.

Parametry:

1. **@ModuleID** (`INT`)
   ID modułu, do którego ma zostać dodane spotkanie.

2. **@MeetingType** (`NVARCHAR(20)`)
   Typ spotkania, może być "Stationary" (stacjonarne) lub "Online".
3. **@StartDate** (`DATETIME`)
   Data rozpoczęcia spotkania.
4. **@EndDate** (`DATETIME`)
   Data zakończenia spotkania.
5. **@RoomID** (`INT`, opcjonalne)
   ID sali, jeżeli spotkanie jest stacjonarne.

Walidacje:

- Sprawdza, czy podany `@ModuleID` istnieje w tabeli `coursemodules`.
- Upewnia się, że data rozpoczęcia spotkania jest wcześniejsza niż data zakończenia.
- Weryfikuje, czy typ spotkania jest odpowiedni do typu modułu (stacjonarne spotkania tylko dla modułów stacjonarnych lub hybrydowych, online tylko dla online lub hybrydowych).
- W przypadku spotkania stacjonarnego wymaga podania `@RoomID`.

```
CREATE PROCEDURE Addcoursemeeting @ModuleID    INT,
                                  @MeetingType NVARCHAR(20),
                                  @StartDate   DATETIME,
                                  @EndDate     DATETIME,
                                  @RoomID      INT = NULL
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          DECLARE @ModuleType INT;

          SELECT @ModuleType = moduletype
          FROM   coursemodules
          WHERE  moduleid = @ModuleID;

          IF @ModuleType IS NULL
            BEGIN
                THROW 50001, 'Nie znaleziono modułu o podanym ModuleID.',
1;
            END;

          IF @StartDate >= @EndDate
            BEGIN
                THROW 50002,
      'Data rozpoczęcia musi być wcześniejsza niż data zakończenia.', 1
      ;
            END;
```

```
          IF @MeetingType = 'Stationary'
            BEGIN
                IF @ModuleType IN ( 1, 4 )
                  BEGIN
                      THROW 50003,
'Do modułu o typie innym niż Stacjonarny lub Hybrydowy nie można dodać
spotkania Stacjonarnego.'
    , 1;
END;

    IF @RoomID IS NULL
      BEGIN
          THROW 50004, 'RoomID jest wymagany dla stacjonarnych spotkań.',
1;
      END;

    INSERT INTO stationarycoursemeeting
                (moduleid,
                 startdate,
                 enddate,
                 roomid)
    VALUES      (@ModuleID,
                 @StartDate,
                 @EndDate,
                 @RoomID);
END
ELSE IF @MeetingType = 'Online'
  BEGIN
      IF @ModuleType NOT IN ( 3, 4 )
        BEGIN
            THROW 50005,
'Do modułu o typie innym niż Online lub Hybrydowy nie można dodać
spotkania Online.'
    , 1;
END;

    INSERT INTO onlinecoursemeeting
                (moduleid,
                 startdate,
                 enddate)
    VALUES      (@ModuleID,
                 @StartDate,
                 @EndDate);
END;
ELSE
  BEGIN
      THROW 50006,
```

```
      'Nieprawidłowy typ spotkania. Użyj "Stationary" lub "Online".', 1;
  END

    COMMIT TRANSACTION;
END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        PRINT 'Wystąpił błąd.';

        PRINT Error_message();
    END catch
END;
```

AddLanguage

J.K.

Cel:
Procedura dodaje nowy język do tabeli `languages` w bazie danych. Sprawdza, czy język o danej nazwie już istnieje.

Parametry:

1. **@LanguageName** (`NVARCHAR(100)`)
   Nazwa języka, który ma zostać dodany.

Walidacje:

- Sprawdza, czy język o danej nazwie już istnieje w tabeli `languages`.

```
CREATE PROCEDURE Addlanguage @LanguageName NVARCHAR(100)
AS
  BEGIN
      BEGIN try
          IF EXISTS (SELECT 1
                     FROM   languages
                     WHERE  languagename = @LanguageName)
            BEGIN
                PRINT 'Błąd: Język o podanej nazwie już istnieje.';

                RETURN;
            END

          INSERT INTO languages
```

```
                    (languagename)
          VALUES    (@LanguageName);

          PRINT 'Język został pomyślnie dodany.';
      END try

      BEGIN catch
          PRINT 'Wystąpił błąd podczas dodawania języka: '
                + Error_message();
      END catch
  END;
```

UpdateLanguage

J.K.

Cel:
Procedura umożliwia zmianę nazwy istniejącego języka w tabeli `languages`. Sprawdza, czy język o danym `@LanguageID` istnieje oraz czy nowa nazwa nie jest już używana.

Parametry:

1. **@LanguageID** (`INT`)
   ID języka, który ma zostać zaktualizowany.
2. **@NewLanguageName** (`NVARCHAR(100)`)
   Nowa nazwa języka.

Walidacje:

- Sprawdza, czy język o podanym `@LanguageID` istnieje w tabeli `languages`.
- Upewnia się, że nowa nazwa języka (`@NewLanguageName`) nie jest już używana.

```
CREATE PROCEDURE Updatelanguage @LanguageID      INT,
                                @NewLanguageName NVARCHAR(100)
AS
  BEGIN
      BEGIN try
          IF NOT EXISTS (SELECT 1
                         FROM   languages
                         WHERE  languageid = @LanguageID)
            BEGIN
                PRINT 'Błąd: Język o podanym ID nie istnieje.';

                RETURN;
            END

          IF EXISTS (SELECT 1
```

```
                    FROM   languages
                    WHERE  languagename = @NewLanguageName)
          BEGIN
              PRINT 'Błąd: Język o podanej nowej nazwie już istnieje.';

              RETURN;
          END

        UPDATE languages
        SET    languagename = @NewLanguageName
        WHERE  languageid = @LanguageID;

        PRINT 'Nazwa języka została pomyślnie zaktualizowana.';
    END try

    BEGIN catch
        PRINT 'Wystąpił błąd podczas aktualizacji języka: '
              + Error_message();
    END catch
END;
```

DeleteLanguageFromTranslatedLanguage

J.K.

Cel:
Procedura usuwa rekordy w tabeli `translatedlanguage` związane z podanym językiem. Sama tabela `languages` pozostaje bez zmian.

Parametry:

1. **@LanguageID** (`INT`)
   ID języka, którego rekordy mają zostać usunięte z tabeli `translatedlanguage`.

Walidacje:

- Sprawdza, czy język o podanym `@LanguageID` istnieje w tabeli `languages`.
- Informuje, jeśli język nie został znaleziony.

```
CREATE PROCEDURE Deletelanguagefromtranslatedlanguage @LanguageID INT
AS
  BEGIN
      IF EXISTS (SELECT 1
                 FROM   languages
                 WHERE  languageid = @LanguageID)
        BEGIN
            DELETE FROM translatedlanguage
```

```
            WHERE  languageid = @LanguageID;

            PRINT
'Rekordy w tabeli TranslatedLanguage zostały usunięte, ale język pozostaje
w tabeli Languages.'
    ;
END
ELSE
  BEGIN
      PRINT 'Nie znaleziono języka o podanym LanguageID.';
  END
END;

CREATE PROCEDURE Addcountry @CountryName NVARCHAR(100)
AS
  BEGIN
      BEGIN try
          IF EXISTS (SELECT 1
                     FROM   countries
                     WHERE  countryname = @CountryName)
            BEGIN
                PRINT 'Błąd: Kraj o podanej nazwie już istnieje.';

                RETURN;
            END

          INSERT INTO countries
                      (countryname)
          VALUES      (@CountryName);

          PRINT 'Kraj został pomyślnie dodany.';
      END try

      BEGIN catch
          PRINT 'Wystąpił błąd podczas dodawania kraju: '
                + Error_message();
      END catch
  END;
```

AddCountry

J.K.

Cel:
Procedura dodaje nowy kraj do tabeli `countries`. Sprawdza, czy kraj o podanej nazwie już istnieje.

Parametry:

1. **@CountryName** (`NVARCHAR(100)`)
   Nazwa kraju, który ma zostać dodany.

Walidacje:

- Sprawdza, czy kraj o podanej nazwie już istnieje w tabeli `countries`.

```
CREATE PROCEDURE Updatecountry @CountryID      INT,
                               @NewCountryName NVARCHAR(100)
AS
  BEGIN
      BEGIN try
          IF NOT EXISTS (SELECT 1
                         FROM   countries
                         WHERE  countryid = @CountryID)
            BEGIN
                PRINT 'Błąd: Kraj o podanym ID nie istnieje.';

                RETURN;
            END

          IF EXISTS (SELECT 1
                     FROM   countries
                     WHERE  countryname = @NewCountryName)
            BEGIN
                PRINT 'Błąd: Kraj o podanej nowej nazwie już istnieje.';

                RETURN;
            END

          UPDATE countries
          SET    countryname = @NewCountryName
          WHERE  countryid = @CountryID;

          PRINT 'Nazwa kraju została pomyślnie zaktualizowana.';
      END try

      BEGIN catch
          PRINT 'Wystąpił błąd podczas aktualizacji kraju: '
                + Error_message();
      END catch
  END;
```

UpdateCountry

J.K.

Cel:
Procedura umożliwia aktualizację nazwy istniejącego kraju w tabeli `countries`.

Parametry:

1. **@CountryID** (`INT`)
   ID kraju, który ma zostać zaktualizowany.
2. **@NewCountryName** (`NVARCHAR(100)`)
   Nowa nazwa kraju.

Walidacje:

- Sprawdza, czy kraj o podanym `@CountryID` istnieje.
- Upewnia się, że nowa nazwa kraju nie jest już używana.

```
CREATE PROCEDURE Addcoursemodule @CourseID     INT,
                                 @ModuleName   NVARCHAR(255),
                                 @ModuleType   INT,
                                 @LecturerID   INT,
                                 @TranslatorID INT = NULL,
                                 @LanguageID   INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF @LanguageID NOT IN (SELECT languageid
                                 FROM   translatedlanguage)
            BEGIN
                THROW 67007, 'Language not available.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   formofactivity
                         WHERE  activitytypeid = @ModuleType)
            BEGIN
                THROW 50001, 'Nieprawidłowy typ modułu.', 1;
            END;

          INSERT INTO coursemodules
                      (courseid,
                       modulename,
```

```
                        moduletype,
                        lecturerid,
                        translatorid,
                        languageid)
            VALUES      (@CourseID,
                        @ModuleName,
                        @ModuleType,
                        @LecturerID,
                        @TranslatorID,
                        @LanguageID);

            COMMIT TRANSACTION;
        END try

        BEGIN catch
            ROLLBACK TRANSACTION;

            THROW;
        END catch
    END;
```

AddCity

J.K.

Cel:
Procedura dodaje nowe miasto do tabeli `cities`. Sprawdza, czy miasto o podanej nazwie i kodzie pocztowym już istnieje w wybranym kraju.

Parametry:

1. **@CityName** (`NVARCHAR(100)`)
   Nazwa miasta.
2. **@PostalCode** (`NVARCHAR(6)`)
   Kod pocztowy miasta.
3. **@CountryID** (`INT`)
   ID kraju, w którym miasto ma zostać dodane.

Walidacje:

- Sprawdza, czy kraj o podanym `@CountryID` istnieje.
- Weryfikuje, czy miasto o podanej nazwie i kodzie pocztowym nie istnieje już w danym kraju.

```
CREATE PROCEDURE Addcity @CityName   NVARCHAR(100),
                         @PostalCode NVARCHAR(6),
                         @CountryID  INT
```

```
AS
  BEGIN
      BEGIN try
          IF NOT EXISTS (SELECT 1
                         FROM   countries
                         WHERE  countryid = @CountryID)
            BEGIN
                PRINT 'Błąd: Kraj o podanym ID nie istnieje.';

                RETURN;
            END

          IF EXISTS (SELECT 1
                     FROM   cities
                     WHERE  cityname = @CityName
                            AND postalcode = @PostalCode
                            AND countryid = @CountryID)
            BEGIN
                PRINT
'Błąd: Miasto o podanej nazwie i kodzie pocztowym już istnieje w wybranym
kraju.'
    ;

    RETURN;
END

    INSERT INTO cities
                (cityname,
                 postalcode,
                 countryid)
    VALUES      (@CityName,
                 @PostalCode,
                 @CountryID);

    PRINT 'Miasto zostało pomyślnie dodane.';
END try

    BEGIN catch
        PRINT 'Wystąpił błąd podczas dodawania miasta: '
              + Error_message();
    END catch
END;
```

UpdateCity

J.K.

Cel:
Procedura umożliwia aktualizację nazwy miasta lub kodu pocztowego w tabeli `cities`.

Parametry:

1. **@CityID** (`INT`)
   ID miasta, które ma zostać zaktualizowane.
2. **@NewCityName** (`NVARCHAR(100)`, opcjonalne)
   Nowa nazwa miasta.
3. **@NewPostalCode** (`NVARCHAR(20)`, opcjonalne)
   Nowy kod pocztowy miasta.

Walidacje:

- Sprawdza, czy miasto o podanym `@CityID` istnieje.
- Weryfikuje, czy nowa nazwa miasta (jeśli podana) nie jest już używana w danym kraju przez inne miasto.

```
CREATE PROCEDURE Updatecity @CityID        INT,
                            @NewCityName   NVARCHAR(100) = NULL,
                            @NewPostalCode NVARCHAR(20) = NULL
AS
  BEGIN
      IF EXISTS (SELECT 1
                 FROM   cities
                 WHERE  cityid = @CityID)
        BEGIN
            DECLARE @CountryID INT;

            SELECT @CountryID = countryid
            FROM   cities
            WHERE  cityid = @CityID;

            IF @NewCityName IS NOT NULL
               AND EXISTS (SELECT 1
                           FROM   cities
                           WHERE  cityname = @NewCityName
                                  AND countryid = @CountryID
                                  AND cityid <> @CityID)
              BEGIN
                  PRINT
'Miasto o tej nazwie już istnieje w wybranym kraju. Aktualizacja nie może
zostać wykonana.'
    ;
END
ELSE
  BEGIN
```

```
      UPDATE cities
      SET    cityname = Isnull(@NewCityName, cityname),
             postalcode = Isnull(@NewPostalCode, postalcode)
      WHERE  cityid = @CityID;

      PRINT 'Dane miasta zostały zaktualizowane.';
  END
END
ELSE
  BEGIN
      PRINT 'Miasto o podanym CityID nie zostało znalezione.';
  END
END;
```

AddUser

O.B.

Cel:
Dodaje nowego użytkownika do tabeli `users`. Sprawdza, czy miasto podane w parametrach istnieje.

Parametry:

1. @**FirstName** (`NVARCHAR(40)`) - Imię użytkownika.
2. @**LastName** (`NVARCHAR(40)`) - Nazwisko użytkownika.
3. @**Street** (`NVARCHAR(40)`) - Adres użytkownika.
4. @**City** (`NVARCHAR(40)`) - Miasto użytkownika.
5. @**Email** (`NVARCHAR(40)`) - Adres e-mail użytkownika.
6. @**Phone** (`NVARCHAR(40)`) - Numer telefonu użytkownika.
7. @**DateOfBirth** (`DATE`) - Data urodzenia użytkownika.

Walidacje:

- Sprawdza, czy miasto istnieje w tabeli `cities`.

```
CREATE PROCEDURE Adduser @FirstName   NVARCHAR(40),
                         @LastName    NVARCHAR(40),
                         @Street      NVARCHAR(40),
                         @City        NVARCHAR(40),
                         @Email       NVARCHAR(40),
                         @Phone       NVARCHAR(40),
                         @DateOfBirth DATE
AS
  BEGIN
      BEGIN try
          DECLARE @CityID INT;
```

```
            SELECT @CityID = cityid
            FROM   cities
            WHERE  cityname = @City;

            IF @CityID IS NULL
              BEGIN
                  THROW 50001, 'City does not exist in the Cities table.',
1;
              END;

            INSERT INTO users
                        (firstname,
                         lastname,
                         street,
                         cityid,
                         email,
                         phone,
                         dateofbirth)
            VALUES      (@FirstName,
                         @LastName,
                         @Street,
                         @CityID,
                         @Email,
                         @Phone,
                         @DateOfBirth);
        END try

        BEGIN catch
            PRINT 'Wystąpił błąd.';

            PRINT Error_message();
        END catch
    END;
```

AddEmployee

O.B.

Cel:
Dodaje użytkownika jako pracownika w tabeli `employees` oraz przypisuje mu rolę.

Parametry:

1. **@EmployeeID** (`INT`) - ID użytkownika będącego pracownikiem.
2. **@HireDate** (`DATE`) - Data zatrudnienia.
3. **@DegreeName** (`NVARCHAR(50)`, opcjonalne) - Tytuł naukowy pracownika.

4. **@RoleName** (`NVARCHAR(50)`) - Rola przypisana pracownikowi.

Walidacje:

- Sprawdza, czy użytkownik istnieje w tabeli `users`.
- Weryfikuje istnienie roli i tytułu naukowego w odpowiednich tabelach.

```
CREATE PROCEDURE Addemployee @EmployeeID INT,
                             @HireDate   DATE,
                             @DegreeName NVARCHAR(50) = NULL,
                             @RoleName   NVARCHAR(50)
AS
  BEGIN
      BEGIN try
          IF NOT EXISTS (SELECT 1
                         FROM   users
                         WHERE  userid = @EmployeeID)
            BEGIN
                THROW 50002, 'User does not exist in the Users table.', 1;
            END;

          DECLARE @RoleID INT;

          SELECT @RoleID = roleid
          FROM   roles
          WHERE  rolename = @RoleName;

          IF @RoleID IS NULL
            BEGIN
                THROW 50005, 'Role does not exist in the Roles table.', 1;
            END;

          DECLARE @DegreeID INT = NULL;

          IF @DegreeName IS NOT NULL
            BEGIN
                SELECT @DegreeID = degreeid
                FROM   degrees
                WHERE  degreename = @DegreeName;

                IF @DegreeID IS NULL
                  BEGIN
                      THROW 50006, 'Degree does not exist in the Degrees
table.'
                      ,
                      1;
                  END;
```

```
            END;

        INSERT INTO employees
                    (employeeid,
                     hiredate,
                     degreeid)
        VALUES      (@EmployeeID,
                     @HireDate,
                     @DegreeID);

        EXEC Adduserrole
          @UserID = @EmployeeID,
          @RoleID = @RoleID;
    END try

    BEGIN catch
        PRINT 'Wystąpił błąd.';

        PRINT Error_message();
    END catch
END;
```

DeleteUser

O.B.

Cel:
Usuwa użytkownika z systemu, w tym jego dane z tabel powiązanych (`employees` i `usersroles`).

Parametry:

1. **@UserID** (`INT`) - ID użytkownika do usunięcia.

Walidacje:

- Sprawdza, czy użytkownik istnieje w tabeli `users`.
- Usuwa dane pracownika, jeśli użytkownik jest również w tabeli `employees`.

```
CREATE PROCEDURE Deleteuser @UserID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   users
```

```
                    WHERE  userid = @UserID)
          BEGIN
              THROW 60001, 'User does not exist.', 1;
          END;

          IF EXISTS (SELECT 1
                     FROM   employees
                     WHERE  employeeid = @UserID)
          BEGIN
              DELETE FROM employees
              WHERE  employeeid = @UserID;
          END;

          DELETE FROM usersroles
          WHERE  userid = @UserID;

          DELETE FROM users
          WHERE  userid = @UserID;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          PRINT 'Wystąpił błąd.';

          PRINT Error_message();
      END catch
  END;
```

DeleteWebinar

J.K.

Cel:
Usuwa webinar z tabeli `webinars`.

Parametry:

1. **@WebinarID** (`INT`) - ID webinaru do usunięcia.

Walidacje:

- Sprawdza, czy webinar istnieje w tabeli `webinars`.

```
CREATE PROCEDURE Deletewebinar @WebinarID INT
```

```
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   webinars
                         WHERE  webinarid = @WebinarID)
            BEGIN
                THROW 60002, 'Webinar does not exist.', 1;
            END;

          DELETE FROM webinars
          WHERE  webinarid = @WebinarID;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          PRINT 'Wystąpił błąd.';

          PRINT Error_message();
      END catch
  END;
```

DeleteStudyMeeting

O.B.

Cel:
Usuwa spotkanie studenckie wraz z powiązanymi danymi z tabel `stationarymeetings` i `onlinemeetings`.

Parametry:

1. **@MeetingID** (`INT`) - ID spotkania do usunięcia.

Walidacje:

- Sprawdza, czy spotkanie istnieje w tabeli `studymeetings`.

```
CREATE PROCEDURE Deletestudymeeting @MeetingID INT
AS
  BEGIN
      BEGIN try
```

```
        BEGIN TRANSACTION;

        IF NOT EXISTS (SELECT 1
                       FROM   studymeetings
                       WHERE  meetingid = @MeetingID)
          BEGIN
              THROW 60003, 'Study meeting does not exist.', 1;
          END;

        IF EXISTS (SELECT 1
                   FROM   stationarymeetings
                   WHERE  meetingid = @MeetingID)
          BEGIN
              DELETE FROM stationarymeetings
              WHERE  meetingid = @MeetingID;
          END;

        IF EXISTS (SELECT 1
                   FROM   onlinemeetings
                   WHERE  meetingid = @MeetingID)
          BEGIN
              DELETE FROM onlinemeetings
              WHERE  meetingid = @MeetingID;
          END;

        DELETE FROM studymeetings
        WHERE  meetingid = @MeetingID;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        PRINT 'Wystąpił błąd.';

        PRINT Error_message();
    END catch
END;
```

DeleteSubject

O.B.

Cel:
Usuwa przedmiot z tabeli `subjects` oraz wszystkie powiązane z nim spotkania.

Parametry:

1. @**SubjectID** (`INT`) - ID przedmiotu do usunięcia.

Walidacje:

- Sprawdza, czy przedmiot istnieje w tabeli `subjects`.
- Usuwa powiązane spotkania z tabel `studymeetings`, `stationarymeetings`, i `onlinemeetings`.

```
CREATE PROCEDURE Deletesubject @SubjectID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   subjects
                         WHERE  subjectid = @SubjectID)
            BEGIN
                THROW 60004, 'Subject does not exist.', 1;
            END;

          DECLARE @MeetingID INT;
          DECLARE meetingcursor CURSOR FOR
            SELECT meetingid
            FROM   studymeetings
            WHERE  subjectid = @SubjectID;

          OPEN meetingcursor;

          FETCH next FROM meetingcursor INTO @MeetingID;

          WHILE @@FETCH_STATUS = 0
            BEGIN
                DELETE FROM stationarymeetings
                WHERE  meetingid = @MeetingID;

                DELETE FROM onlinemeetings
                WHERE  meetingid = @MeetingID;

                DELETE FROM studymeetings
                WHERE  meetingid = @MeetingID;

                FETCH next FROM meetingcursor INTO @MeetingID;
            END;
```

```
        CLOSE meetingcursor;

        DEALLOCATE meetingcursor;

        DELETE FROM subjects
        WHERE  subjectid = @SubjectID;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch
END;
```

DeleteStudy

O.B.

Cel:
Usuwa kierunek studiów z tabeli `studies` oraz wszystkie powiązane z nim przedmioty.

Parametry:

1. **@StudyID** (`INT`) - ID kierunku studiów do usunięcia.

Walidacje:

- Sprawdza, czy kierunek studiów istnieje w tabeli `studies`.
- Usuwa wszystkie przedmioty i ich powiązania korzystając z procedury `DeleteSubject`.

```
CREATE PROCEDURE Deletestudy @StudyID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   studies
                         WHERE  studiesid = @StudyID)
            BEGIN
                THROW 60005, 'Study does not exist.', 1;
            END;
```

```
        DECLARE @SubjectID INT;
        DECLARE subjectcursor CURSOR FOR
          SELECT subjectid
          FROM   subjects
          WHERE  studiesid = @StudyID;

        OPEN subjectcursor;

        FETCH next FROM subjectcursor INTO @SubjectID;

        WHILE @@FETCH_STATUS = 0
          BEGIN
              EXEC Deletesubject
                @SubjectID = @SubjectID;

              FETCH next FROM subjectcursor INTO @SubjectID;
          END;

        CLOSE subjectcursor;

        DEALLOCATE subjectcursor;

        DELETE FROM studies
        WHERE  studiesid = @StudyID;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch
END;
```

DeleteCourseMeeting

O.B.

- **Cel**: Usunięcie spotkania kursowego z tabel `onlinecoursemeeting` oraz `stationarycoursemeeting` na podstawie identyfikatora spotkania.
- **Parametry**:
  - `@MeetingID INT`: Identyfikator spotkania, które ma zostać usunięte.
- **Walidacje**:
  - Brak walidacji, ponieważ procedura bezpośrednio usuwa dane z tabel na podstawie przekazanego identyfikatora.

```
CREATE PROCEDURE Deletecoursemeeting @MeetingID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          DELETE FROM onlinecoursemeeting
          WHERE  meetingid = @MeetingID;

          DELETE FROM stationarycoursemeeting
          WHERE  meetingid = @MeetingID;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          THROW;
      END catch
  END;
```

DeleteCourseModule

O.B.

- **Cel**: Usunięcie modułu kursowego z tabeli `coursemodule` na podstawie identyfikatora modułu.
- **Parametry**:
  - `@ModuleID INT`: Identyfikator modułu, który ma zostać usunięty.
- **Walidacje**:
  - Brak walidacji istnienia modułu, zakłada się, że identyfikator jest poprawny.

```
CREATE PROCEDURE Deletecoursemodule @ModuleID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          DELETE FROM coursemodules
          WHERE  moduleid = @ModuleID;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;
```

```
            THROW;
        END catch
    END;
```

DeleteCourse

O.B.

- **Cel**: Usunięcie kursu oraz wszystkich powiązanych modułów.
- **Parametry**:
    - `@CourseID INT`: Identyfikator kursu, który ma zostać usunięty.
- **Walidacje**:
    - Sprawdzenie, czy kurs o podanym `CourseID` istnieje w tabeli `courses`. Jeśli nie istnieje, procedura generuje błąd.

```
CREATE PROCEDURE Deletecourse @CourseID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   courses
                         WHERE  courseid = @CourseID)
            BEGIN
                THROW 60008, 'Course does not exist.', 1;
            END;

          DECLARE @ModuleID INT;
          DECLARE modulecursor CURSOR FOR
            SELECT moduleid
            FROM   coursemodules
            WHERE  courseid = @CourseID;

          OPEN modulecursor;

          FETCH next FROM modulecursor INTO @ModuleID;

          WHILE @@FETCH_STATUS = 0
            BEGIN
                EXEC Deletecoursemodule
                  @ModuleID = @ModuleID;

                FETCH next FROM modulecursor INTO @ModuleID;
            END;
```

```
        CLOSE modulecursor;

        DEALLOCATE modulecursor;

        DELETE FROM courses
        WHERE  courseid = @CourseID;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch
END;
```

DeleteEmployee

O.B.

- **Cel**: Usunięcie pracownika z tabeli `employees` wraz z jego rolami (z wyjątkiem administratora).
- **Parametry**:
  - `@EmployeeID INT`: Identyfikator pracownika, który ma zostać usunięty.
- **Walidacje**:
  - Sprawdzenie, czy pracownik o podanym `EmployeeID` istnieje w tabeli `employees`. Jeśli nie istnieje, procedura generuje błąd.

```
CREATE PROCEDURE Deleteemployee @EmployeeID INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @EmployeeID)
            BEGIN
                THROW 60009, 'Employee does not exist.', 1;
            END;

          DECLARE @UserID INT;

          SELECT @UserID = employeeid
          FROM   employees
```

```
            WHERE  employeeid = @EmployeeID;

            DELETE FROM usersroles
            WHERE  userid = @UserID
                   AND roleid != 1;

            DELETE FROM employees
            WHERE  employeeid = @EmployeeID;

            COMMIT TRANSACTION;
        END try

        BEGIN catch
            ROLLBACK TRANSACTION;

            THROW;
        END catch
    END;
```

UpdateUser

O.B.

- **Cel**: Aktualizacja danych użytkownika w tabeli `users`.
- **Parametry**:
  - `@UserID INT`: Identyfikator użytkownika.
  - `@FirstName VARCHAR(40)`: Imię użytkownika.
  - `@LastName VARCHAR(40)`: Nazwisko użytkownika.
  - `@Street VARCHAR(40)`: Ulica zamieszkania.
  - `@City VARCHAR(40)`: Miasto zamieszkania.
  - `@Email VARCHAR(40)`: Adres e-mail.
  - `@Phone VARCHAR(40)`: Numer telefonu.
  - `@DateOfBirth DATE`: Data urodzenia.
- **Walidacje**:
  - Sprawdzenie, czy użytkownik o podanym `UserID` istnieje.
  - Sprawdzenie, czy miasto o podanej nazwie istnieje w tabeli `cities`.

```
CREATE PROCEDURE Updateuser @UserID      INT,
                            @FirstName   VARCHAR(40),
                            @LastName    VARCHAR(40),
                            @Street      VARCHAR(40),
                            @City        VARCHAR(40),
                            @Email       VARCHAR(40),
                            @Phone       VARCHAR(40),
                            @DateOfBirth DATE
```

```
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   users
                         WHERE  userid = @UserID)
            BEGIN
                THROW 60010, 'User does not exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   cities
                         WHERE  cityname = @City)
            BEGIN
                THROW 60011, 'City does not exist.', 1;
            END;

          UPDATE users
          SET    firstname = @FirstName,
                 lastname = @LastName,
                 street = @Street,
                 cityid = (SELECT cityid
                           FROM   cities
                           WHERE  cityname = @City),
                 email = @Email,
                 phone = @Phone,
                 dateofbirth = @DateOfBirth
          WHERE  userid = @UserID;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          THROW;
      END catch
  END;
```

UpdateEmployee

O.B.

- **Cel**: Aktualizacja szczegółowych danych pracownika, takich jak data zatrudnienia i stopień naukowy.

- **Parametry**:
  - `@EmployeeID INT`: Identyfikator pracownika.
  - `@HireDate DATE`: Data zatrudnienia.
  - `@DegreeName VARCHAR(50)`: Nazwa stopnia naukowego (opcjonalne).
- **Walidacje**:
  - Sprawdzenie, czy pracownik o podanym `EmployeeID` istnieje.
  - Jeśli podano `DegreeName`, weryfikacja, czy stopień naukowy istnieje w tabeli `degrees`.

```
CREATE PROCEDURE Updateemployee @EmployeeID INT,
                                @HireDate   DATE,
                                @DegreeName VARCHAR(50) = NULL
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @EmployeeID)
            BEGIN
                THROW 60012, 'Employee does not exist.', 1;
            END;

          DECLARE @DegreeID INT = NULL;

          IF @DegreeName IS NOT NULL
            BEGIN
                SELECT @DegreeID = degreeid
                FROM   degrees
                WHERE  degreename = @DegreeName;

                IF @DegreeID IS NULL
                  BEGIN
                      THROW 60013, 'Degree does not exist.', 1;
                  END;
            END;

          UPDATE employees
          SET    hiredate = @HireDate,
                 degreeid = @DegreeID
          WHERE  employeeid = @EmployeeID;

          COMMIT TRANSACTION;
      END try
```

```
        BEGIN catch
            ROLLBACK TRANSACTION;

            THROW;
        END catch
    END;
```

UpdateCourse

O.B.

- **Cel**: Aktualizacja szczegółów kursu, takich jak nazwa, koordynator, opis, cena i limit studentów.
- **Parametry**:
    - `@CourseID INT`: Identyfikator kursu.
    - `@CourseName VARCHAR(40)`: Nazwa kursu.
    - `@CourseCoordinatorID INT`: Identyfikator koordynatora kursu.
    - `@CourseDescription VARCHAR(255)`: Opis kursu.
    - `@CoursePrice MONEY`: Cena kursu.
    - `@StudentLimit INT`: Limit studentów (opcjonalne).
- **Walidacje**:
    - Sprawdzenie, czy kurs o podanym `CourseID` istnieje.
    - Sprawdzenie, czy koordynator kursu o podanym `CourseCoordinatorID` istnieje i ma rolę przypisaną do koordynatora (`roleid = 3`).

```
CREATE PROCEDURE Updatecourse @CourseID            INT,
                              @CourseName          VARCHAR(40),
                              @CourseCoordinatorID INT,
                              @CourseDescription   VARCHAR(255),
                              @CoursePrice         MONEY,
                              @StudentLimit        INT = NULL
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   courses
                         WHERE  courseid = @CourseID)
            BEGIN
                THROW 61001, 'Course does not exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   employees E
                                JOIN usersroles UR
```

```
                                ON E.employeeid = UR.userid
                        WHERE  E.employeeid = @CourseCoordinatorID
                               AND UR.roleid = 3)
            BEGIN
                THROW 61002,
      'CourseCoordinatorID is not valid or does not have the required
role.'
      , 1;
            END;

          UPDATE courses
          SET    coursename = @CourseName,
                 coursecoordinatorid = @CourseCoordinatorID,
                 coursedescription = @CourseDescription,
                 courseprice = @CoursePrice,
                 studentlimit = @StudentLimit
          WHERE  courseid = @CourseID;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          THROW;
      END catch
  END;
```

UpdateWebinar

J.K.

- **Cel**: Aktualizacja szczegółów webinaru, takich jak nazwa, nauczyciel, tłumacz, język, terminy oraz odnośniki.
- **Parametry**:
  - `@WebinarID INT`: Identyfikator webinaru.
  - `@WebinarName VARCHAR(40)`: Nazwa webinaru.
  - `@WebinarDescription VARCHAR(255)`: Opis webinaru.
  - `@TeacherID INT`: Identyfikator nauczyciela prowadzącego.
  - `@Price MONEY`: Cena webinaru.
  - `@LanguageID INT`: Identyfikator języka.
  - `@TranslatorID INT`: Identyfikator tłumacza (opcjonalne).
  - `@StartDate DATETIME`: Data rozpoczęcia.
  - `@EndDate DATETIME`: Data zakończenia.
  - `@VideoLink VARCHAR(255)`: Link do wideo (opcjonalne).
  - `@MeetingLink VARCHAR(255)`: Link do spotkania (opcjonalne).

- **Walidacje**:
  - Sprawdzenie, czy webinar o podanym `WebinarID` istnieje.
  - Sprawdzenie, czy nauczyciel o podanym `TeacherID` posiada odpowiednią rolę (`roleid = 8`).
  - Weryfikacja, czy język o podanym `LanguageID` istnieje.
  - Jeśli podano `TranslatorID`, weryfikacja, czy tłumacz istnieje i ma rolę przypisaną do tłumacza (`roleid = 2`).

```
CREATE PROCEDURE Updatewebinar @WebinarID          INT,
                               @WebinarName        VARCHAR(40),
                               @WebinarDescription VARCHAR(255),
                               @TeacherID          INT,
                               @Price              MONEY,
                               @LanguageID         INT,
                               @TranslatorID       INT = NULL,
                               @StartDate          DATETIME,
                               @EndDate            DATETIME,
                               @VideoLink          VARCHAR(255) = NULL,
                               @MeetingLink        VARCHAR(255) = NULL
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   webinars
                         WHERE  webinarid = @WebinarID)
            BEGIN
                THROW 62001, 'Webinar does not exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   usersroles
                         WHERE  userid = @TeacherID
                                AND roleid = 8)
            BEGIN
                THROW 62002,
      'Invalid TeacherID. The user is not assigned as a webinar
lecturer.',
      1;
            END;

          IF @LanguageID NOT IN (SELECT languageid
                                 FROM   translatedlanguage)
            BEGIN
                THROW 67007, 'Language not available.', 1;
```

```
            END;

        IF @TranslatorID IS NOT NULL
           AND NOT EXISTS (SELECT 1
                           FROM   usersroles
                           WHERE  userid = @TranslatorID
                                  AND roleid = 2)
            BEGIN
                THROW 62003,
      'Invalid TranslatorID. The user is not assigned as a translator.',
      1;
            END;

        UPDATE webinars
        SET    webinarname = @WebinarName,
               webinardescription = @WebinarDescription,
               teacherid = @TeacherID,
               price = @Price,
               languageid = @LanguageID,
               translatorid = @TranslatorID,
               startdate = @StartDate,
               enddate = @EndDate,
               videolink = @VideoLink,
               meetinglink = @MeetingLink
        WHERE  webinarid = @WebinarID;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch;
END;

go
```

UpdateCourseModule

O.B.

- **Cel**: Aktualizacja danych modułu kursowego, takich jak typ modułu, język, tłumacz, wykładowca.
- **Parametry**:
    - `@ModuleID INT`: Identyfikator modułu.

    - `@CourseID INT`: Identyfikator kursu.
    - `@ModuleName VARCHAR(255)`: Nazwa modułu.
    - `@ModuleType INT`: Typ modułu.
    - `@LecturerID INT`: Identyfikator wykładowcy.
    - `@LanguageID INT`: Identyfikator języka.
    - `@TranslatorID INT`: Identyfikator tłumacza (opcjonalne).
- **Walidacje**:
    - Sprawdzenie, czy moduł o podanym `ModuleID` istnieje.
    - Sprawdzenie, czy wykładowca o podanym `LecturerID` ma przypisaną rolę wykładowcy (`roleid = 6`).
    - Weryfikacja, czy język o podanym `LanguageID` istnieje.
    - Jeśli podano `TranslatorID`, weryfikacja, czy tłumacz istnieje i ma rolę przypisaną do tłumacza (`roleid = 2`).
    - Walidacja zmiany typu modułu (brak konfliktów z istniejącymi spotkaniami online/stacjonarnymi).

```
CREATE PROCEDURE Updatecoursemodule @ModuleID     INT,
                                    @CourseID     INT,
                                    @ModuleName   VARCHAR(255),
                                    @ModuleType   INT,
                                    @LecturerID   INT,
                                    @LanguageID   INT,
                                    @TranslatorID INT = NULL
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   coursemodules
                         WHERE  moduleid = @ModuleID)
            BEGIN
                THROW 63001, 'Module does not exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   usersroles
                         WHERE  userid = @LecturerID
                                AND roleid = 6)
            BEGIN
                THROW 63002,
                'Invalid LecturerID. The user is not assigned as a
lecturer.'
                ,
                1;
```

```
            END;

        IF @LanguageID NOT IN (SELECT languageid
                               FROM   translatedlanguage)
            BEGIN
                THROW 67007, 'Language not available.', 1;
            END;

        IF @TranslatorID IS NOT NULL
           AND NOT EXISTS (SELECT 1
                           FROM   usersroles
                           WHERE  userid = @TranslatorID
                                  AND roleid = 2)
            BEGIN
                THROW 63003,
      'Invalid TranslatorID. The user is not assigned as a translator.',
      1;
            END;

        IF @ModuleType = 1
           AND EXISTS (SELECT 1
                       FROM   onlinecoursemeeting
                       WHERE  moduleid = @ModuleID)
            BEGIN
                THROW 63004,
'Cannot change module type to Stationary. Online meetings exist for this
module.'
    , 1;
END;

    IF @ModuleType = 3
       AND EXISTS (SELECT 1
                   FROM   stationarycoursemeeting
                   WHERE  moduleid = @ModuleID)
      BEGIN
          THROW 63005,
'Cannot change module type to Online. Stationary meetings exist for this
module.'
    , 1;
END;

    UPDATE coursemodules
    SET    courseid = @CourseID,
           modulename = @ModuleName,
           moduletype = @ModuleType,
           lecturerid = @LecturerID,
           languageid = @LanguageID,
```

```
            translatorid = @TranslatorID
    WHERE   moduleid = @ModuleID;

    COMMIT TRANSACTION;
END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch;
END;

go
```

UpdateCourseModuleMeeting

O.B.

- **Cel**: Aktualizacja spotkania modułu kursowego (stacjonarnego lub online) z uwzględnieniem typu spotkania.
- **Parametry**:
  - `@MeetingID INT`: Identyfikator spotkania.
  - `@ModuleID INT`: Identyfikator modułu kursowego.
  - `@MeetingType INT`: Typ spotkania (`1` - stacjonarne, `2` - online).
  - `@StartDateTime DATETIME`: Data i godzina rozpoczęcia.
  - `@EndDateTime DATETIME`: Data i godzina zakończenia.
  - `@LocationOrLink VARCHAR(255)`: Lokalizacja lub link do spotkania.
- **Walidacje**:
  - Sprawdzenie istnienia modułu kursowego w tabeli `coursemodule`.
  - Weryfikacja, czy typ spotkania jest zgodny z typem modułu kursowego.
  - Sprawdzenie istnienia spotkania w odpowiedniej tabeli (`stationarycoursemeeting` lub `onlinecoursemeeting`).

```
CREATE PROCEDURE Updatecoursemodulemeeting @MeetingID      INT,
                                          @ModuleID       INT,
                                          @MeetingType    INT,
                                          @StartDateTime  DATETIME,
                                          @EndDateTime    DATETIME,
                                          @LocationOrLink VARCHAR(255)
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;
```

```
            IF NOT EXISTS (SELECT 1
                           FROM   coursemodules
                           WHERE  moduleid = @ModuleID)
              BEGIN
                  THROW 64001, 'Module does not exist.', 1;
              END;

            DECLARE @ModuleType INT;

            SELECT @ModuleType = moduletype
            FROM   coursemodules
            WHERE  moduleid = @ModuleID;

            IF @MeetingType = 1
               AND @ModuleType NOT IN ( 1, 4 )
              BEGIN
                  THROW 64002,
                  'Cannot add a stationary meeting to a non-stationary
module.'
                  ,
                  1;
              END;

            IF @MeetingType = 2
               AND @ModuleType NOT IN ( 3, 4 )
              BEGIN
                  THROW 64003,
                  'Cannot add an online meeting to a non-online module.', 1
                  ;
              END;

            IF @MeetingType = 1
               AND NOT EXISTS (SELECT 1
                               FROM   stationarycoursemeeting
                               WHERE  meetingid = @MeetingID
                                      AND moduleid = @ModuleID)
              BEGIN
                  THROW 64004,
                  'Stationary meeting does not exist for the specified
module.'
                  ,
                  1;
              END;

            IF @MeetingType = 2
               AND NOT EXISTS (SELECT 1
                               FROM   onlinecoursemeeting
```

```
                        WHERE  meetingid = @MeetingID
                               AND moduleid = @ModuleID)
          BEGIN
              THROW 64005,
              'Online meeting does not exist for the specified module.'
              , 1;
          END;

        IF @MeetingType = 1
          BEGIN
              UPDATE stationarycoursemeeting
              SET    startdate = @StartDateTime,
                     enddate = @EndDateTime
              WHERE  meetingid = @MeetingID
                     AND moduleid = @ModuleID;
          END
        ELSE IF @MeetingType = 2
          BEGIN
              UPDATE onlinecoursemeeting
              SET    startdate = @StartDateTime,
                     enddate = @EndDateTime,
                     meetinglink = @LocationOrLink
              WHERE  meetingid = @MeetingID
                     AND moduleid = @ModuleID;
          END;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          THROW;
      END catch;
  END;

go
```

UpdateStudies

O.B.

- **Cel**: Aktualizacja szczegółów studiów, takich jak nazwa, koordynator, cena, liczba semestrów i limit studentów.
- **Parametry**:
  - `@StudiesID INT`: Identyfikator studiów.
  - `@StudiesCoordinatorID INT`: Identyfikator koordynatora studiów.

    - `@StudyName VARCHAR(40)`: Nazwa studiów.
    - `@StudyPrice MONEY`: Cena studiów.
    - `@StudyDescription VARCHAR(255)`: Opis studiów.
    - `@NumberOfTerms INT`: Liczba semestrów.
    - `@StudentLimit INT`: Limit studentów.
- **Walidacje**:
    - Sprawdzenie istnienia studiów w tabeli `studies`.
    - Sprawdzenie istnienia koordynatora w tabeli `employees` oraz jego roli jako koordynatora studiów (`roleid = 4`).

```
CREATE PROCEDURE Updatestudies @StudiesID            INT,
                               @StudiesCoordinatorID INT,
                               @StudyName            VARCHAR(40),
                               @StudyPrice           MONEY,
                               @StudyDescription     VARCHAR(255),
                               @NumberOfTerms        INT,
                               @StudentLimit         INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   studies
                         WHERE  studiesid = @StudiesID)
            BEGIN
                THROW 65001, 'Studies with the specified ID do not
exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @StudiesCoordinatorID)
            BEGIN
                THROW 65002, 'The specified coordinator does not exist.',
1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   usersroles
                         WHERE  userid = (SELECT userid
                                          FROM   employees
                                          WHERE
                                 employeeid = @StudiesCoordinatorID)
                                 AND roleid = 4)
            BEGIN
```

```
                THROW 65003,
'The specified coordinator does not have the required role (Coordinator of
Studies).'
    , 1;
END;

    UPDATE studies
    SET    studiescoordinatorid = @StudiesCoordinatorID,
           studyname = @StudyName,
           studyprice = @StudyPrice,
           studydescription = @StudyDescription,
           numberofterms = @NumberOfTerms,
           studentlimit = @StudentLimit
    WHERE  studiesid = @StudiesID;

    COMMIT TRANSACTION;
END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch;
END;

go
```

UpdateSubject

O.B.

- **Cel**: Aktualizacja przedmiotu, w tym jego nazwy, opisu, powiązanych studiów i wykładowcy.
- **Parametry**:
  - `@SubjectID INT`: Identyfikator przedmiotu.
  - `@StudiesID INT`: Identyfikator studiów.
  - `@SubjectName VARCHAR(40)`: Nazwa przedmiotu.
  - `@SubjectDescription VARCHAR(255)`: Opis przedmiotu.
  - `@LecturerID INT`: Identyfikator wykładowcy.
- **Walidacje**:
  - Sprawdzenie istnienia przedmiotu w tabeli `subjects`.
  - Weryfikacja istnienia studiów w tabeli `studies`.
  - Sprawdzenie istnienia wykładowcy w tabeli `employees` oraz jego roli jako wykładowcy (`roleid = 6`).

```
CREATE PROCEDURE Updatesubject @SubjectID          INT,
```

```
                              @StudiesID          INT,
                              @SubjectName        VARCHAR(40),
                              @SubjectDescription VARCHAR(255),
                              @LecturerID         INT
AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   subjects
                         WHERE  subjectid = @SubjectID)
            BEGIN
                THROW 66001, 'Subject with the specified ID does not
exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   studies
                         WHERE  studiesid = @StudiesID)
            BEGIN
                THROW 66002, 'Studies with the specified ID do not
exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   employees
                         WHERE  employeeid = @LecturerID)
            BEGIN
                THROW 66003, 'The specified lecturer does not exist.', 1;
            END;

          IF NOT EXISTS (SELECT 1
                         FROM   usersroles
                         WHERE  userid = (SELECT userid
                                          FROM   employees
                                          WHERE  employeeid = @LecturerID)
                                AND roleid = 6)
            BEGIN
                THROW 66004,
      'The specified lecturer does not have the required role
(Lecturer).',
      1;
            END

          UPDATE subjects
          SET    studiesid = @StudiesID,
```

```
                subjectname = @SubjectName,
                subjectdescription = @SubjectDescription,
                teacherid = @LecturerID
        WHERE   subjectid = @SubjectID;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch;
END;

go
```

UpdateStudyMeeting

O.B.

- **Cel**: Aktualizacja szczegółów spotkania studiów (stacjonarnego lub online), w tym lokalizacji, limitu studentów i tłumacza.
- **Parametry**:
  - `@MeetingID INT`: Identyfikator spotkania.
  - `@MeetingType INT`: Typ spotkania (`1` - stacjonarne, `2` - online).
  - `@SubjectID INT`: Identyfikator przedmiotu.
  - `@LecturerID INT`: Identyfikator wykładowcy.
  - `@MeetingPrice MONEY`: Cena spotkania.
  - `@MeetingPriceForOthers MONEY`: Cena spotkania dla innych.
  - `@StartTime DATETIME`: Czas rozpoczęcia spotkania.
  - `@EndTime DATETIME`: Czas zakończenia spotkania.
  - `@LanguageID INT`: Identyfikator języka.
  - `@TranslatorID INT`: Identyfikator tłumacza (opcjonalnie).
  - `@MeetingLink VARCHAR(255)`: Link do spotkania online (opcjonalnie).
  - `@StudentLimit INT`: Limit studentów.
  - `@RoomID INT`: Identyfikator sali (opcjonalnie).
- **Walidacje**:
  - Sprawdzenie istnienia spotkania w tabeli `studymeetings`.
  - Weryfikacja poprawności typu spotkania.
  - Weryfikacja istnienia języka w tabeli `translatedlanguage`.
  - Wymóg podania `RoomID` dla spotkań stacjonarnych i `MeetingLink` dla online.

```
CREATE PROCEDURE Updatestudymeeting @MeetingID             INT,
```

```
                                    @MeetingType           INT,
                                    @SubjectID             INT,
                                    @LecturerID            INT,
                                    @MeetingPrice          MONEY,
                                    @MeetingPriceForOthers MONEY,
                                    @StartTime             DATETIME,
                                    @EndTime               DATETIME,
                                    @LanguageID            INT,
                                    @TranslatorID          INT = NULL,
                                    @MeetingLink           VARCHAR(255) =
NULL,
                                    @StudentLimit          INT,
                                    @RoomID                INT = NULL

AS
  BEGIN
      BEGIN try
          BEGIN TRANSACTION;

          IF NOT EXISTS (SELECT 1
                         FROM   studymeetings
                         WHERE  meetingid = @MeetingID)
            BEGIN
                THROW 67001,
                'Study meeting with the specified ID does not exist.'
                , 1;
            END;

          IF @MeetingType NOT IN ( 1, 2 )
            BEGIN
                THROW 67002,
                'Invalid MeetingType. Must be 1 (Stationary) or 2
(Online).',
                1
                ;
            END;

          IF @LanguageID NOT IN (SELECT languageid
                                 FROM   translatedlanguage)
            BEGIN
                THROW 67007, 'Language not available.', 1;
            END;

          UPDATE studymeetings
          SET    subjectid = @SubjectID,
                 lecturerid = @LecturerID,
                 meetingtype = @MeetingType,
                 meetingprice = @MeetingPrice,
```

```
        meetingpriceforothers = @MeetingPriceForOthers,
        starttime = @StartTime,
        endtime = @EndTime,
        languageid = @LanguageID,
        translatorid = @TranslatorID
WHERE   meetingid = @MeetingID;

IF @MeetingType = 1
  BEGIN
      IF @RoomID IS NULL
        BEGIN
            THROW 67003,
            'RoomID must be provided for stationary meetings.',
            1;
        END;

      DELETE FROM onlinemeetings
      WHERE  meetingid = @MeetingID;

      IF EXISTS (SELECT 1
                 FROM   stationarymeetings
                 WHERE  meetingid = @MeetingID)
        BEGIN
            UPDATE stationarymeetings
            SET    roomid = @RoomID,
                   studentlimit = @StudentLimit
            WHERE  meetingid = @MeetingID;
        END
      ELSE
        BEGIN
            INSERT INTO stationarymeetings
                        (meetingid,
                         roomid,
                         studentlimit)
            VALUES      (@MeetingID,
                         @RoomID,
                         @StudentLimit);
        END;
  END
ELSE IF @MeetingType = 2
  BEGIN
      IF @MeetingLink IS NULL
        BEGIN
            THROW 67004,
            'MeetingLink must be provided for online meetings.'
            , 1;
        END;
```

```
                DELETE FROM stationarymeetings
                WHERE  meetingid = @MeetingID;

                IF EXISTS (SELECT 1
                           FROM   onlinemeetings
                           WHERE  meetingid = @MeetingID)
                  BEGIN
                      UPDATE onlinemeetings
                      SET    meetinglink = @MeetingLink,
                             studentlimit = @StudentLimit
                      WHERE  meetingid = @MeetingID;
                  END
                ELSE
                  BEGIN
                      INSERT INTO onlinemeetings
                                  (meetingid,
                                   meetinglink,
                                   studentlimit)
                      VALUES      (@MeetingID,
                                   @MeetingLink,
                                   @StudentLimit);
                  END;
            END;

          COMMIT TRANSACTION;
      END try

      BEGIN catch
          ROLLBACK TRANSACTION;

          THROW;
      END catch;
  END;

go
```

AddUserWithRoleAndEmployee

K.B.

- **Cel**: Dodanie użytkownika z przypisaną rolą oraz ewentualnie jako pracownika z datą zatrudnienia i stopniem naukowym.
- **Parametry**:
  - `@FirstName VARCHAR(40)`: Imię użytkownika.
  - `@LastName VARCHAR(40)`: Nazwisko użytkownika.

    - `@Street VARCHAR(40)`: Ulica zamieszkania.
    - `@CityName VARCHAR(40)`: Miasto zamieszkania.
    - `@Email VARCHAR(40)`: Adres e-mail.
    - `@Phone VARCHAR(40)`: Numer telefonu.
    - `@DateOfBirth DATE`: Data urodzenia.
    - `@RoleName VARCHAR(40)`: Nazwa roli użytkownika.
    - `@HireDate DATE`: Data zatrudnienia (opcjonalnie).
    - `@DegreeName VARCHAR(255)`: Stopień naukowy (opcjonalnie).
- **Walidacje**:
    - Sprawdzenie istnienia miasta w tabeli `cities`.
    - Weryfikacja unikalności adresu e-mail w tabeli `users`.
    - Sprawdzenie istnienia roli w tabeli `roles`.
    - Walidacja stopnia naukowego, jeśli został podany.

```
CREATE PROCEDURE Adduserwithroleandemployee (@FirstName   VARCHAR(40),
                                             @LastName    VARCHAR(40),
                                             @Street      VARCHAR(40),
                                             @CityName    VARCHAR(40),
                                             @Email       VARCHAR(40),
                                             @Phone       VARCHAR(40),
                                             @DateOfBirth DATE,
                                             @RoleName    VARCHAR(40),
                                             @HireDate    DATE = NULL,
                                             @DegreeName  VARCHAR(255) =
NULL)
AS
  BEGIN
      SET nocount ON;

      BEGIN try
          BEGIN TRANSACTION;

          DECLARE @CityID INT;

          SELECT @CityID = cityid
          FROM   cities
          WHERE  cityname = @CityName;

          IF @CityID IS NULL
            BEGIN
                THROW 50000, 'Podane miasto nie istnieje w tabeli
Cities.', 1;
            END;

          IF EXISTS (SELECT 1
                     FROM   users
```

```
                    WHERE  email = @Email)
            BEGIN
                THROW 50003, 'Podany adres e-mail już istnieje w tabeli
Users.'
                ,
                1;
            END;

          DECLARE @UserID INT;

          INSERT INTO users
                      (firstname,
                       lastname,
                       street,
                       cityid,
                       email,
                       phone,
                       dateofbirth)
          VALUES      (@FirstName,
                       @LastName,
                       @Street,
                       @CityID,
                       @Email,
                       @Phone,
                       @DateOfBirth);

          SET @UserID = Scope_identity();

          DECLARE @RoleID INT;

          SELECT @RoleID = roleid
          FROM   roles
          WHERE  rolename = @RoleName;

          IF @RoleID IS NULL
            BEGIN
                THROW 50001, 'Podana rola nie istnieje w tabeli Roles.',
1;
            END;

          EXEC Adduserrole
            @UserID,
            @RoleID;

          IF @RoleID <> 1
            BEGIN
                DECLARE @DegreeID INT = NULL;
```

```
            IF @DegreeName IS NOT NULL
    BEGIN
        SELECT @DegreeID = degreeid
        FROM   degrees
        WHERE  degreename = @DegreeName;

        IF @DegreeID IS NULL
          BEGIN
              THROW 50002,
              'Podany stopień naukowy nie istnieje w tabeli Degrees.',
              1;
          END;
    END;

              INSERT INTO employees
                          (employeeid,
                           hiredate,
                           degreeid)
              VALUES      (@UserID,
                           @HireDate,
                           @DegreeID);
          END;

        COMMIT TRANSACTION;
    END try

    BEGIN catch
        ROLLBACK TRANSACTION;

        THROW;
    END catch;
END;

go
```

AddTranslationLanguage

J.K.

- **Cel**: Dodanie tłumacza do języka.
- **Parametry**:
  - `@TranslatorID INT`: Identyfikator tłumacza.
  - `@LanguageID INT`: Identyfikator języka.
- **Walidacje**:
  - Sprawdzenie, czy tłumacz ma odpowiednią rolę (`roleid = 2`).

- Sprawdzenie istnienia języka w tabeli `languages`.
- Weryfikacja, czy rekord nie istnieje już w tabeli `translatedlanguage`.

```
CREATE PROCEDURE Addtranslatedlanguage @TranslatorID INT,
                                       @LanguageID   INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   usersroles
                     WHERE  userid = @TranslatorID
                            AND roleid = 2)
        BEGIN
            PRINT 'Użytkownik nie jest tłumaczem.';

            RETURN;
        END;

      IF NOT EXISTS (SELECT 1
                     FROM   languages
                     WHERE  languageid = @LanguageID)
        BEGIN
            PRINT 'Podany język nie istnieje.';

            RETURN;
        END;

      IF EXISTS (SELECT 1
                 FROM   translatedlanguage
                 WHERE  translatorid = @TranslatorID
                        AND languageid = @LanguageID)
        BEGIN
            PRINT 'Rekord już istnieje.';

            RETURN;
        END;

      INSERT INTO translatedlanguage
                  (translatorid,
                   languageid)
      VALUES      (@TranslatorID,
                   @LanguageID);

      PRINT 'Rekord został dodany.';
  END;
```

UpdateTranslatedLanguage

J.K.

- **Cel**: Aktualizacja tłumacza oraz przypisanego języka w tabeli `translatedlanguage`.
- **Parametry**:
  - `@OldTranslatorID INT`: ID tłumacza, który ma zostać zmieniony.
  - `@OldLanguageID INT`: ID języka, który ma zostać zmieniony.
  - `@NewTranslatorID INT`: ID nowego tłumacza.
  - `@NewLanguageID INT`: ID nowego języka.
- **Walidacje**:
  - Sprawdzenie, czy istnieje rekord do aktualizacji (tłumacz i język).
  - Sprawdzenie, czy nowy tłumacz ma odpowiednią rolę (ID roli 2).
  - Sprawdzenie, czy nowy język istnieje w tabeli `languages`.
  - Sprawdzenie, czy rekord z nowym tłumaczem i językiem już istnieje w tabeli `translatedlanguage`.

```
CREATE PROCEDURE Updatetranslatedlanguage @OldTranslatorID INT,
                                          @OldLanguageID   INT,
                                          @NewTranslatorID INT,
                                          @NewLanguageID   INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   translatedlanguage
                     WHERE  translatorid = @OldTranslatorID
                            AND languageid = @OldLanguageID)
        BEGIN
            PRINT 'Rekord do modyfikacji nie istnieje.';

            RETURN;
        END;

      IF NOT EXISTS (SELECT 1
                     FROM   usersroles
                     WHERE  userid = @NewTranslatorID
                            AND roleid = 2)
        BEGIN
            PRINT 'Nowy tłumacz nie ma odpowiedniej roli.';

            RETURN;
        END;

      IF NOT EXISTS (SELECT 1
                     FROM   languages
                     WHERE  languageid = @NewLanguageID)
        BEGIN
```

```
            PRINT 'Podany język nie istnieje.';

            RETURN;
        END;

    IF EXISTS (SELECT 1
               FROM   translatedlanguage
               WHERE  translatorid = @NewTranslatorID
                      AND languageid = @NewLanguageID)
        BEGIN
            PRINT 'Nowy rekord już istnieje.';

            RETURN;
        END;

    UPDATE translatedlanguage
    SET    translatorid = @NewTranslatorID,
           languageid = @NewLanguageID
    WHERE  translatorid = @OldTranslatorID
           AND languageid = @OldLanguageID;

    PRINT 'Rekord został zaktualizowany.';
  END;
```

DeleteTranslatedLanguage

J.K.

- **Cel**: Usunięcie powiązania tłumacza z językiem w tabeli `translatedlanguage`.
- **Parametry**:
  - `@TranslatorID INT`: ID tłumacza.
  - `@LanguageID INT`: ID języka.
- **Walidacje**:
  - Sprawdzenie, czy istnieje rekord do usunięcia (tłumacz i język).

```
CREATE PROCEDURE Deletetranslatedlanguage @TranslatorID INT,
                                          @LanguageID   INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   translatedlanguage
                     WHERE  translatorid = @TranslatorID
                            AND languageid = @LanguageID)
        BEGIN
            PRINT 'Rekord do usunięcia nie istnieje.';

            RETURN;
```

```
    END;

  DELETE FROM translatedlanguage
  WHERE  translatorid = @TranslatorID
         AND languageid = @LanguageID;

  PRINT 'Rekord został usunięty.';
END;
```

AddRoom

J.K.

- **Cel**: Dodanie nowej sali do bazy danych.
- **Parametry**:
  - `@RoomName NVARCHAR(100)`: Nazwa sali.
  - `@Street NVARCHAR(100)`: Ulica, na której znajduje się sala.
  - `@CityID INT`: ID miasta.
  - `@Limit INT`: Limit osób w sali.
- **Walidacje**:
  - Sprawdzenie, czy sala o podanej nazwie już istnieje w danym mieście.
  - Sprawdzenie, czy limit osób w sali jest większy niż 0.

```
CREATE PROCEDURE Addroom @RoomName NVARCHAR(100),
                         @Street   NVARCHAR(100),
                         @CityID   INT,
                         @Limit    INT
AS
  BEGIN
      IF EXISTS (SELECT 1
                 FROM   rooms
                 WHERE  roomname = @RoomName
                        AND cityid = @CityID)
        BEGIN
            PRINT 'Sala o podanej nazwie już istnieje w tym mieście.';

            RETURN;
        END;

      IF @Limit <= 0
        BEGIN
            PRINT 'Limit osób musi być większy niż 0.';

            RETURN;
        END;

      INSERT INTO rooms
```

```
                (roomname,
                 street,
                 cityid,
                 limit)
      VALUES    (@RoomName,
                 @Street,
                 @CityID,
                 @Limit);

      PRINT 'Sala została dodana.';
  END;
```

UpdateRoom

J.K.

- **Cel**: Aktualizacja danych sali (zmiana nazwy).
- **Parametry**:
  - `@OldRoomID INT`: ID sali do modyfikacji.
  - `@NewRoomName NVARCHAR(100)`: Nowa nazwa sali.
- **Walidacje**:
  - Sprawdzenie, czy sala do modyfikacji istnieje.
  - Sprawdzenie, czy sala o nowej nazwie już nie istnieje w tym samym mieście.

```
CREATE PROCEDURE Updateroom @OldRoomID   INT,
                            @NewRoomName NVARCHAR(100)
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   rooms
                     WHERE  roomid = @OldRoomID)
        BEGIN
            PRINT 'Sala do modyfikacji nie istnieje.';

            RETURN;
        END;

      IF EXISTS (SELECT 1
                 FROM   rooms
                 WHERE  roomname = @NewRoomName
                        AND roomid != @OldRoomID)
        BEGIN
            PRINT 'Sala o podanej nazwie już istnieje w tym mieście.';

            RETURN;
        END;
```

```
    UPDATE rooms
    SET    roomname = @NewRoomName
    WHERE  roomid = @OldRoomID;

    PRINT 'Sala została zaktualizowana.';
  END;
```

AddInternship

J.K.

- **Cel**: Dodanie praktyk dla studentów w określonym terminie.
- **Parametry**:
  - `@StudiesID INT`: ID kierunku studiów.
  - `@InternshipCoordinatorID INT`: ID koordynatora praktyk.
  - `@StartDate DATE`: Data rozpoczęcia praktyk.
  - `@EndDate DATE`: Data zakończenia praktyk.
  - `@NumberOfHours INT`: Liczba godzin praktyk.
  - `@Term INT`: Numer terminu praktyk.
- **Walidacje**:
  - Sprawdzenie, czy numer terminu jest prawidłowy (mieszczący się w zakresie dostępnych terminów dla danego kierunku).
  - Sprawdzenie, czy praktyki w danym terminie już istnieją.
  - Sprawdzenie, czy praktyki trwają dokładnie 14 dni.
  - Sprawdzenie, czy koordynator praktyk ma odpowiednią rolę (ID roli 4).
  - Sprawdzenie, czy praktyki nie kolidują z zajęciami dydaktycznymi.

```
CREATE PROCEDURE Addinternship @StudiesID               INT,
                               @InternshipCoordinatorID INT,
                               @StartDate               DATE,
                               @EndDate                 DATE,
                               @NumberOfHours           INT,
                               @Term                    INT
AS
  BEGIN
      DECLARE @NumberOfTerms INT;

      SELECT @NumberOfTerms = numberofterms
      FROM   studies
      WHERE  studiesid = @StudiesID;

      IF @Term < 1
          OR @Term > @NumberOfTerms
        BEGIN
            PRINT 'Invalid Term for the given Studies.';
```

```
            RETURN;
        END

    IF EXISTS (SELECT 1
               FROM   internship
               WHERE  studiesid = @StudiesID
                      AND term = @Term)
        BEGIN
            PRINT
        'There is already an internship for this term of these studies.'
            ;

            RETURN;
        END

    IF Datediff(day, @StartDate, @EndDate) <> 14
        BEGIN
            PRINT 'Internship must last exactly 14 days.';

            RETURN;
        END

    IF NOT EXISTS (SELECT 1
                   FROM   usersroles
                   WHERE  userid = @InternshipCoordinatorID
                          AND roleid = 4)
        BEGIN
            PRINT 'The specified user is not a coordinator for
internships.';

            RETURN;
        END

    IF EXISTS (SELECT 1
               FROM   studymeetings SM
                      INNER JOIN subjects S
                              ON SM.subjectid = S.subjectid
               WHERE  S.studiesid = @StudiesID
                      AND ( ( @StartDate BETWEEN SM.starttime AND
SM.endtime )
                             OR ( @EndDate BETWEEN SM.starttime AND
                            SM.endtime ) ))
        BEGIN
            PRINT
        'Internship overlaps with a study meeting for the given studies.';

            RETURN;
```

```
        END

    INSERT INTO internship
                (studiesid,
                 internshipcoordinatorid,
                 startdate,
                 enddate,
                 numerofhours,
                 term)
    VALUES      (@StudiesID,
                 @InternshipCoordinatorID,
                 @StartDate,
                 @EndDate,
                 @NumberOfHours,
                 @Term);

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

UpdateInternship

J.K.

- **Cel**: Aktualizacja danych praktyk studenckich.
- **Parametry**:
  - `@InternshipID INT`: ID praktyk do modyfikacji.
  - `@StudiesID INT`: ID kierunku studiów.
  - `@InternshipCoordinatorID INT`: ID koordynatora praktyk.
  - `@StartDate DATE`: Data rozpoczęcia praktyk.
  - `@EndDate DATE`: Data zakończenia praktyk.
  - `@NumberOfHours INT`: Liczba godzin praktyk.
  - `@Term INT`: Numer terminu praktyk.
- **Walidacje**:
  - Sprawdzenie, czy numer terminu jest prawidłowy (mieszczący się w zakresie dostępnych terminów dla danego kierunku).
  - Sprawdzenie, czy praktyki w danym terminie już istnieją (z wyjątkiem aktualizowanego rekordu).
  - Sprawdzenie, czy praktyki trwają dokładnie 14 dni.
  - Sprawdzenie, czy koordynator praktyk ma odpowiednią rolę (ID roli 4).
  - Sprawdzenie, czy praktyki nie kolidują z zajęciami dydaktycznymi.

```
CREATE PROCEDURE Updateinternship @InternshipID            INT,
                                  @StudiesID               INT,
                                  @InternshipCoordinatorID INT,
                                  @StartDate               DATE,
                                  @EndDate                 DATE,
```

```
                                    @NumberOfHours          INT,
                                    @Term                   INT
AS
  BEGIN
      DECLARE @NumberOfTerms INT;

      SELECT @NumberOfTerms = numberofterms
      FROM   studies
      WHERE  studiesid = @StudiesID;

      IF @Term < 1
          OR @Term > @NumberOfTerms
        BEGIN
            PRINT 'Invalid Term for the given Studies.';

            RETURN;
        END

      IF EXISTS (SELECT 1
                 FROM   internship
                 WHERE  studiesid = @StudiesID
                        AND term = @Term
                        AND internshipid != @InternshipID)
        BEGIN
            PRINT
        'There is already an internship for this term of these studies.'
            ;

            RETURN;
        END

      IF Datediff(day, @StartDate, @EndDate) <> 14
        BEGIN
            PRINT 'Internship must last exactly 14 days.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   usersroles
                     WHERE  userid = @InternshipCoordinatorID
                            AND roleid = 4)
        BEGIN
            PRINT 'The specified user is not a coordinator for
internships.';

            RETURN;
```

```
        END

      IF EXISTS (SELECT 1
                 FROM   studymeetings SM
                        INNER JOIN subjects S
                                ON SM.subjectid = S.subjectid
                 WHERE  S.studiesid = @StudiesID
                        AND ( ( @StartDate BETWEEN SM.starttime AND
SM.endtime )
                                OR ( @EndDate BETWEEN SM.starttime AND
                            SM.endtime ) ))
        BEGIN
            PRINT
        'Internship overlaps with a study meeting for the given studies.';

            RETURN;
        END;

      UPDATE internship
      SET    studiesid = @StudiesID,
             internshipcoordinatorid = @InternshipCoordinatorID,
             startdate = @StartDate,
             enddate = @EndDate,
             numerofhours = @NumberOfHours,
             term = @Term
      WHERE  internshipid = @InternshipID;

      PRINT 'Internship updated successfully.';
  END;
```

DeleteInternship

J.K.

- **Cel**: Usunięcie praktyk o określonym ID.
- **Parametry**:
  - `@InternshipID INT`: ID praktyk do usunięcia.
- **Walidacje**:
  - Sprawdzenie, czy praktyki o podanym ID istnieją w tabeli `internship`.
  - Jeśli praktyki nie istnieją, wyświetl komunikat "Internship not found."
  - Jeśli praktyki istnieją, usuwanie rekordu z tabeli `internship`.

```
CREATE PROCEDURE Deleteinternship @InternshipID INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   internship
```

```
                    WHERE  internshipid = @InternshipID)
        BEGIN
            PRINT 'Internship not found.';

            RETURN;
        END

      DELETE FROM internship
      WHERE  internshipid = @InternshipID;

      PRINT 'Internship deleted successfully.';
  END;
```

AddCourseModulePassed

J.K.

- **Cel**: Dodanie rekordu o zdanym module przez studenta.
- **Parametry**:
  - `@ModuleID INT`: ID modułu kursu.
  - `@StudentID INT`: ID studenta.
  - `@Passed BIT`: Status zdania modułu (1 = zaliczone, 0 = niezaliczone).
- **Walidacje**:
  - Sprawdzenie, czy moduł o podanym `ModuleID` istnieje w tabeli `coursemodule`.
  - Sprawdzenie, czy student o podanym `StudentID` istnieje w tabeli `users` i ma rolę studenta.
  - Sprawdzenie, czy student jest zapisany na kurs i ma dostęp do modułu.
  - Sprawdzenie, czy rekord o tym samym module i studencie już istnieje w tabeli `coursemodulepassed`.
  - Jeśli wszystkie walidacje są pozytywne, dodanie rekordu do tabeli `coursemodulepassed`.

```
CREATE PROCEDURE Addcoursemodulepassed @ModuleID  INT,
                                       @StudentID INT,
                                       @Passed    BIT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   coursemodules
                     WHERE  moduleid = @ModuleID)
        BEGIN
            PRINT 'Moduł o podanym ID nie istnieje.'

            RETURN
        END
```

```
    IF NOT EXISTS (SELECT 1
                   FROM   usersroles
                   WHERE  roleid = 1
                          AND userid = @StudentID)
      BEGIN
          PRINT 'Student o podanym ID nie istnieje.'

          RETURN
      END

    IF NOT EXISTS (SELECT 1
                   FROM   orders O
                          JOIN orderdetails OD
                            ON O.orderid = OD.orderid
                   WHERE  O.studentid = @StudentID
                          AND OD.typeofactivity = 2
                          AND OD.activityid IN (SELECT courseid
                                                FROM   courses
                                                WHERE
                              courseid = OD.activityid)
                          AND EXISTS (SELECT 1
                                      FROM   coursemodules CM
                                      WHERE  CM.courseid = OD.activityid
                                             AND CM.moduleid =
@ModuleID))
      BEGIN
          PRINT
      'Student nie jest zapisany na kurs lub nie ma dostępu do tego
modułu.'

          RETURN
      END

    IF EXISTS (SELECT 1
               FROM   coursemodulespassed
               WHERE  moduleid = @ModuleID
                      AND studentid = @StudentID)
      BEGIN
          PRINT 'Rekord już istnieje dla tego studenta i modułu.'

          RETURN
      END

    INSERT INTO coursemodulespassed
                (moduleid,
                 studentid,
```

```
                 passed)
      VALUES     (@ModuleID,
                  @StudentID,
                  @Passed)

      PRINT 'Rekord został pomyślnie dodany.'
  END
```

UpdateCourseModulePassed

J.K.

- **Cel**: Aktualizacja rekordu o module zdanym przez studenta.
- **Parametry**:
  - `@ModuleID INT`: ID modułu kursu.
  - `@StudentID INT`: ID studenta.
  - `@Passed BIT`: Nowy status zdania modułu (1 = zaliczone, 0 = niezaliczone).
- **Walidacje**:
  - Sprawdzenie, czy student jest zapisany na kurs i ma dostęp do modułu.
  - Sprawdzenie, czy rekord o tym samym module i studencie istnieje w tabeli `coursemodulepassed`.
  - Jeśli rekord istnieje, wykonanie aktualizacji w tabeli `coursemodulepassed`.

```
CREATE PROCEDURE Updatecoursemodulepassed @ModuleID  INT,
                                          @StudentID INT,
                                          @Passed    BIT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   orders O
                            JOIN orderdetails OD
                              ON O.orderid = OD.orderid
                     WHERE  O.studentid = @StudentID
                            AND OD.typeofactivity = 2
                            AND OD.activityid IN (SELECT courseid
                                                  FROM   courses
                                                  WHERE
                                courseid = OD.activityid)
                            AND EXISTS (SELECT 1
                                        FROM   coursemodules CM
                                        WHERE  CM.courseid = OD.activityid
                                               AND CM.moduleid =
@ModuleID))
          BEGIN
              PRINT
          'Student nie jest zapisany na kurs lub nie ma dostępu do tego
```

```
modułu.'

            RETURN
        END

    IF NOT EXISTS (SELECT 1
                   FROM   coursemodulespassed
                   WHERE  moduleid = @ModuleID
                          AND studentid = @StudentID)
        BEGIN
            PRINT 'Rekord do modyfikacji nie istnieje.'

            RETURN
        END

    UPDATE coursemodulespassed
    SET    passed = @Passed
    WHERE  moduleid = @ModuleID
           AND studentid = @StudentID

    PRINT 'Rekord został pomyślnie zaktualizowany.'
  END
```

DeleteCourseModulePasse

J.K.

- **Cel**: Usunięcie rekordu o module zdanym przez studenta.
- **Parametry**:
  - `@ModuleID INT`: ID modułu kursu.
  - `@StudentID INT`: ID studenta.
- **Walidacje**:
  - Sprawdzenie, czy student jest zapisany na kurs i ma dostęp do modułu.
  - Sprawdzenie, czy rekord o tym samym module i studencie istnieje w tabeli `coursemodulepassed`.
  - Jeśli rekord istnieje, usunięcie go z tabeli `coursemodulepassed`.

```
CREATE PROCEDURE Deletecoursemodulepassed @ModuleID  INT,
                                          @StudentID INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   orders O
                            JOIN orderdetails OD
                              ON O.orderid = OD.orderid
                     WHERE  O.studentid = @StudentID
                            AND OD.typeofactivity = 2
```

```
                            AND OD.activityid IN (SELECT courseid
                                                  FROM   courses
                                                  WHERE
                                courseid = OD.activityid)
                            AND EXISTS (SELECT 1
                                        FROM   coursemodules CM
                                        WHERE  CM.courseid = OD.activityid
                                               AND CM.moduleid =
@ModuleID))
        BEGIN
            PRINT
        'Student nie jest zapisany na kurs lub nie ma dostępu do tego
modułu.'

            RETURN
        END

      IF NOT EXISTS (SELECT 1
                     FROM   coursemodulespassed
                     WHERE  moduleid = @ModuleID
                            AND studentid = @StudentID)
        BEGIN
            PRINT 'Rekord do usunięcia nie istnieje.'

            RETURN
        END

      DELETE FROM coursemodulespassed
      WHERE  moduleid = @ModuleID
             AND studentid = @StudentID

      PRINT 'Rekord został pomyślnie usunięty.'
  END
```

AddUserRole

K.B.

- **Cel**: Dodanie roli do użytkownika.
- **Parametry**:
  - `@UserID INT`: ID użytkownika.
  - `@RoleID INT`: ID roli.
- **Walidacje**:
  - Sprawdzenie, czy użytkownik o podanym `UserID` istnieje w tabeli `users`.
  - Sprawdzenie, czy rola o podanym `RoleID` istnieje w tabeli `roles`.
  - Sprawdzenie, czy użytkownik już posiada tę rolę.

- Jeśli wszystkie walidacje są pozytywne, dodanie roli do użytkownika w tabeli `usersroles`.

```
CREATE PROCEDURE Adduserrole @UserID INT,
                             @RoleID INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   users
                     WHERE  userid = @UserID)
        BEGIN
            PRINT 'Użytkownik o podanym UserID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   roles
                     WHERE  roleid = @RoleID)
        BEGIN
            PRINT 'Rola o podanym RoleID nie istnieje.';

            RETURN;
        END

      IF EXISTS (SELECT 1
                 FROM   usersroles
                 WHERE  userid = @UserID
                        AND roleid = @RoleID)
        BEGIN
            PRINT 'Użytkownik już posiada tę rolę.';

            RETURN;
        END

      INSERT INTO usersroles
                  (userid,
                   roleid)
      VALUES      (@UserID,
                   @RoleID);

      PRINT 'Rola została pomyślnie dodana użytkownikowi.';
  END;
```

AddStudyResult

K.B.

- **Cel**: Dodanie wyniku studiów dla studenta w danym kierunku.
- **Parametry**:
  - `@StudentID INT`: ID studenta.
  - `@StudiesID INT`: ID kierunku studiów.
  - `@GradeID INT`: ID oceny.
- **Walidacje**:
  - Sprawdzenie, czy student o podanym `StudentID` istnieje w tabeli `users`.
  - Sprawdzenie, czy kierunek studiów o podanym `StudiesID` istnieje w tabeli `studies`.
  - Sprawdzenie, czy ocena o podanym `GradeID` istnieje w tabeli `grades`.
  - Sprawdzenie, czy student złożył zamówienie na dany kierunek studiów.
  - Sprawdzenie, czy rekord wyników studiów już istnieje dla tego studenta i kierunku.
  - Jeśli wszystkie walidacje są pozytywne, dodanie wyniku studiów do tabeli `studiesresults`.

```
CREATE PROCEDURE Addstudyresult @StudentID INT,
                                @StudiesID INT,
                                @GradeID   INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   users
                     WHERE  userid = @StudentID)
        BEGIN
            PRINT 'Student o podanym StudentID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   studies
                     WHERE  studiesid = @StudiesID)
        BEGIN
            PRINT 'Kierunek studiów o podanym StudiesID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   grades
                     WHERE  gradeid = @GradeID)
        BEGIN
            PRINT 'Ocena o podanym GradeID nie istnieje.';

            RETURN;
```

```
    END

    IF NOT EXISTS (SELECT 1
                   FROM   orders O
                          INNER JOIN orderdetails OD
                                  ON O.orderid = OD.orderid
                   WHERE  O.studentid = @StudentID
                          AND OD.activityid = @StudiesID)
      BEGIN
          PRINT 'Student nie złożył zamówienia na te studia.';

          RETURN;
      END

    IF EXISTS (SELECT 1
               FROM   studiesresults
               WHERE  studentid = @StudentID
                      AND studiesid = @StudiesID)
      BEGIN
          PRINT 'Rezultat dla tego studenta i kierunku już istnieje.';

          RETURN;
      END

    INSERT INTO studiesresults
                (studentid,
                 studiesid,
                 gradeid)
    VALUES      (@StudentID,
                 @StudiesID,
                 @GradeID);

    PRINT 'Rezultat studiów został pomyślnie dodany.';
END;
```

AddSubjectResult

K.B.

- **Cel**: Dodanie wyniku z przedmiotu dla studenta.
- **Parametry**:
  - `@StudentID INT`: ID studenta.
  - `@SubjectID INT`: ID przedmiotu.
  - `@GradeID INT`: ID oceny.
- **Walidacje**:
  - Sprawdzenie, czy student o podanym `StudentID` istnieje w tabeli `users`.

- Sprawdzenie, czy przedmiot o podanym `SubjectID` istnieje w tabeli `subjects`.
- Sprawdzenie, czy ocena o podanym `GradeID` istnieje w tabeli `grades`.
- Sprawdzenie, czy student złożył zamówienie na studia, w ramach którego przedmiot jest dostępny.
- Sprawdzenie, czy wynik z przedmiotu już istnieje dla tego studenta.
- Jeśli wszystkie walidacje są pozytywne, dodanie wyniku przedmiotu do tabeli `subjectsresults`.

```
CREATE PROCEDURE Addsubjectresult @StudentID INT,
                                  @SubjectID INT,
                                  @GradeID   INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   users
                     WHERE  userid = @StudentID)
        BEGIN
            PRINT 'Student o podanym StudentID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   subjects
                     WHERE  subjectid = @SubjectID)
        BEGIN
            PRINT 'Przedmiot o podanym SubjectID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   grades
                     WHERE  gradeid = @GradeID)
        BEGIN
            PRINT 'Ocena o podanym GradeID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   orders O
                            INNER JOIN orderdetails OD
                                    ON O.orderid = OD.orderid
                            INNER JOIN subjects S
```

```
                                    ON OD.activityid = S.studiesid
                    WHERE  O.studentid = @StudentID
                           AND S.subjectid = @SubjectID)
        BEGIN
            PRINT
'Student nie złożył zamówienia na studia, w ramach którego jest podany
przedmiot.'
    ;

    RETURN;
END

    IF EXISTS (SELECT 1
               FROM   subjectsresults
               WHERE  studentid = @StudentID
                      AND subjectid = @SubjectID)
      BEGIN
          PRINT 'Rezultat dla tego studenta i przedmiotu już istnieje.';

          RETURN;
      END

    INSERT INTO subjectsresults
                (subjectid,
                 studentid,
                 gradeid)
    VALUES      (@SubjectID,
                 @StudentID,
                 @GradeID);

    PRINT 'Rezultat przedmiotu został pomyślnie dodany.';
END;
```

AddinternshipResult

K.B.

Cel: Dodanie wyniku z praktyki dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@InternshipID INT`: ID praktyki.
- `@Passed BIT`: Czy student zdał praktykę (1 = zdał, 0 = nie zdał).

Walidacje:

1. Sprawdzenie, czy student o podanym `StudentID` istnieje w tabeli `users`.
2. Sprawdzenie, czy praktyka o podanym `InternshipID` istnieje w tabeli `internship`.
3. Sprawdzenie, czy student złożył zamówienie na studia, w ramach których odbywają się praktyki.
4. Sprawdzenie, czy wynik z praktyki już istnieje dla tego studenta.

Jeśli wszystkie walidacje są pozytywne, dodanie wyniku z praktyki do tabeli `internshippassed`.

```
CREATE PROCEDURE Addinternshipresult @StudentID    INT,
                                     @InternshipID INT,
                                     @Passed       BIT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   users
                     WHERE  userid = @StudentID)
        BEGIN
            PRINT 'Student o podanym StudentID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   internship
                     WHERE  internshipid = @InternshipID)
        BEGIN
            PRINT 'Praktyka o podanym InternshipID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   orders O
                            INNER JOIN orderdetails OD
                                    ON O.orderid = OD.orderid
                            INNER JOIN internship I
                                    ON OD.activityid = I.studiesid
                     WHERE  O.studentid = @StudentID
                            AND I.internshipid = @InternshipID)
        BEGIN
            PRINT
'Student nie złożył zamówienia na studia, w ramach których odbywają się
praktyki.'
    ;
```

```
    RETURN;
END

    IF EXISTS (SELECT 1
               FROM   internshippassed
               WHERE  studentid = @StudentID
                      AND internshipid = @InternshipID)
      BEGIN
          PRINT 'Rezultat praktyk dla tego studenta już istnieje.';

          RETURN;
      END

    INSERT INTO internshippassed
                (internshipid,
                 studentid,
                 passed)
    VALUES      (@InternshipID,
                 @StudentID,
                 @Passed);

    PRINT 'Rezultat praktyk został pomyślnie dodany.';
END;
```

AddstudyMeetingPresence

K.B.

Cel: Dodanie obecności studenta na spotkaniu.

Parametry:

- `@StudentID INT`: ID studenta.
- `@MeetingID INT`: ID spotkania.
- `@Presence BIT`: Obecność studenta (1 = obecny, 0 = nieobecny).

Walidacje:

1. Sprawdzenie, czy student o podanym `StudentID` istnieje w tabeli `users`.
2. Sprawdzenie, czy spotkanie o podanym `MeetingID` istnieje w tabeli `studymeetings`.
3. Sprawdzenie, czy student wykupił spotkanie, sprawdzając tabele `orders` i `orderdetails`.
4. Sprawdzenie, czy obecność na to spotkanie dla tego studenta już istnieje.

Jeśli wszystkie walidacje są pozytywne, dodanie obecności studenta na spotkaniu do tabeli `studymeetingpresence`.

```
CREATE PROCEDURE Addstudymeetingpresence @StudentID INT,
                                         @MeetingID INT,
                                         @Presence  BIT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   users
                     WHERE  userid = @StudentID)
        BEGIN
            PRINT 'Student o podanym StudentID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   studymeetings
                     WHERE  meetingid = @MeetingID)
        BEGIN
            PRINT 'Spotkanie o podanym MeetingID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   orders O
                            INNER JOIN orderdetails OD
                                    ON O.orderid = OD.orderid
                     WHERE  O.studentid = @StudentID
                            AND OD.activityid = @MeetingID
                     UNION
                     SELECT 1
                     FROM   orders O
                            INNER JOIN orderdetails OD
                                    ON O.orderid = OD.orderid
                            INNER JOIN studymeetingpayment SMP
                                    ON OD.detailid = SMP.detailid
                     WHERE  O.studentid = @StudentID
                            AND SMP.meetingid = @MeetingID)
        BEGIN
            PRINT 'Student nie wykupił tego spotkania.';

            RETURN;
        END

      IF EXISTS (SELECT 1
                 FROM   studymeetingpresence
```

```
                WHERE  studymeetingid = @MeetingID
                       AND studentid = @StudentID)
        BEGIN
            PRINT 'Obecność na to spotkanie dla tego studenta już
istnieje.';

            RETURN;
        END

      INSERT INTO studymeetingpresence
                  (studymeetingid,
                   studentid,
                   presence)
      VALUES      (@MeetingID,
                   @StudentID,
                   @Presence);

      PRINT 'Obecność została pomyślnie dodana.';
  END;
```

AddActivityInsteadOfPresence

K.B.

Cel: Dodanie aktywności, która zastępuje nieobecność na spotkaniu.

Parametry:

- `@StudentID INT`: ID studenta.
- `@MeetingID INT`: ID spotkania.
- `@ActivityID INT`: ID aktywności, która zastępuje nieobecność.
- `@TypeOfActivity INT`: Typ aktywności (np. 1 = webinar, 2 = kurs).

Walidacje:

1. Sprawdzenie, czy student o podanym `StudentID` istnieje w tabeli `users`.
2. Sprawdzenie, czy spotkanie o podanym `MeetingID` istnieje w tabeli `studymeetings`.
3. Sprawdzenie, czy student ma nieobecność na to spotkanie.
4. Sprawdzenie, czy aktywność o podanym `ActivityID` istnieje w tabeli `studymeetings` (lub innych odpowiednich tabelach).
5. Sprawdzenie, czy aktywność została już dodana dla tego studenta i spotkania.

Jeśli wszystkie walidacje są pozytywne, dodanie aktywności w miejsce nieobecności do tabeli `activityinsteadofabsence`.

```
CREATE PROCEDURE Addactivityinsteadofpresence @StudentID      INT,
```

```
                                                @MeetingID      INT,
                                                @ActivityID     INT,
                                                @TypeOfActivity INT
AS
  BEGIN
      IF NOT EXISTS (SELECT 1
                     FROM   users
                     WHERE  userid = @StudentID)
        BEGIN
            PRINT 'Student o podanym StudentID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   studymeetings
                     WHERE  meetingid = @MeetingID)
        BEGIN
            PRINT 'Spotkanie o podanym MeetingID nie istnieje.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   studymeetingpresence
                     WHERE  studentid = @StudentID
                            AND studymeetingid = @MeetingID
                            AND presence = 0)
        BEGIN
            PRINT 'Student nie ma nieobecności na to spotkanie.';

            RETURN;
        END

      IF NOT EXISTS (SELECT 1
                     FROM   studymeetings
                     WHERE  meetingid = @ActivityID)
        BEGIN
            PRINT 'Odrabiana aktywność o podanym ActivityID nie
istnieje.';

            RETURN;
        END

      IF EXISTS (SELECT 1
                 FROM   activityinsteadofabsence
                 WHERE  meetingid = @MeetingID
```

```
                    AND studentid = @StudentID
                    AND activityid = @ActivityID)
      BEGIN
          PRINT
      'Odrabianie dla tego spotkania i studenta zostało już dodane.'
          ;

          RETURN;
      END

    INSERT INTO activityinsteadofabsence
                (meetingid,
                 studentid,
                 activityid,
                 typeofactivity)
    VALUES      (@MeetingID,
                 @StudentID,
                 @ActivityID,
                 @TypeOfActivity);

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

AddOrderWithDetails

K.B.

Cel: Dodanie zamówienia wraz z szczegółami dotyczącymi różnych aktywności.

Parametry:

- `@UserID INT`: ID użytkownika (studenta).
- `@ActivityList ACTIVITYLISTTYPE readonly`: Lista aktywności do zamówienia (przekazywana jako tabela).

Walidacje:

1. Sprawdzenie, czy student o podanym `UserID` istnieje w tabeli `users`.
2. Sprawdzenie, czy wszystkie aktywności z listy istnieją w odpowiednich tabelach (np. `courses`, `studies`, `webinars`, `studymeetings`).
3. Sprawdzenie, czy cena za każdą aktywność została prawidłowo przypisana na podstawie typu aktywności (kurs, studia, spotkanie, webinar).
4. Złożenie zamówienia oraz dodanie szczegółów zamówienia do tabel `orders` i `orderdetails`.

Jeśli wszystkie walidacje są pozytywne, dodanie zamówienia oraz szczegółów aktywności do odpowiednich tabel.

```
CREATE type activitylisttype AS TABLE ( activityid INT, typeofactivity INT
);

CREATE PROCEDURE Addorderwithdetails @UserID       INT,
                                     @ActivityList ACTIVITYLISTTYPE
readonly
AS
  BEGIN
      DECLARE @OrderID INT;
      DECLARE @ActivityID INT;
      DECLARE @TypeOfActivity INT;
      DECLARE @Price DECIMAL(18, 2);
      DECLARE @DetailID INT;
      DECLARE @OrderDate DATETIME = Getdate();
      DECLARE @PaymentLink VARCHAR(255) = NULL;

      INSERT INTO orders
                  (studentid,
                   orderdate)
      VALUES      (@UserID,
                   @OrderDate);

      SET @OrderID = Scope_identity();

      DECLARE activitycursor CURSOR FOR
        SELECT activityid,
               typeofactivity
        FROM   @ActivityList;

      OPEN activitycursor;

      FETCH next FROM activitycursor INTO @ActivityID, @TypeOfActivity;

      WHILE @@FETCH_STATUS = 0
        BEGIN
            IF @TypeOfActivity = 2
              SELECT @Price = courseprice
              FROM   courses
              WHERE  courseid = @ActivityID;
            ELSE IF @TypeOfActivity = 3
              SELECT @Price = studyprice
              FROM   studies
              WHERE  studiesid = @ActivityID;
            ELSE IF @TypeOfActivity = 4
              SELECT @Price = meetingpriceforothers
              FROM   studymeetings
              WHERE  meetingid = @ActivityID;
```

```
ELSE IF @TypeOfActivity = 1
  SELECT @Price = price
  FROM   webinars
  WHERE  webinarid = @ActivityID;

SET @DetailID = Scope_identity();

INSERT INTO orderdetails
            (orderid,
             activityid,
             typeofactivity,
             price,
             paiddate,
             paymentstatus)
VALUES      (@OrderID,
             @ActivityID,
             @TypeOfActivity,
             @Price,
             NULL,
             NULL);

IF @TypeOfActivity = 2
  BEGIN
      DECLARE @AdvancePrice DECIMAL(18, 2) = @Price * 0.10;

      INSERT INTO paymentsadvances
                  (detailid,
                   advanceprice,
                   advancepaiddate,
                   advancepaymentstatus)
      VALUES      (@DetailID,
                   @AdvancePrice,
                   NULL,
                   NULL);
  END

IF @TypeOfActivity = 3
  BEGIN
      DECLARE @MeetingID INT;
      DECLARE @MeetingPrice DECIMAL(18, 2);
      DECLARE meetingcursor CURSOR FOR
        SELECT meetingid,
               meetingprice
        FROM   studymeetings
        WHERE  meetingid = @ActivityID;

      OPEN meetingcursor;
```

```
                    FETCH next FROM meetingcursor INTO @MeetingID,
@MeetingPrice;

                    WHILE @@FETCH_STATUS = 0
                      BEGIN
                          INSERT INTO studymeetingpayment
                                      (detailid,
                                       meetingid,
                                       price,
                                       paiddate,
                                       paymentstatus)
                          VALUES      (@DetailID,
                                       @MeetingID,
                                       @MeetingPrice,
                                       NULL,
                                       NULL);

                          FETCH next FROM meetingcursor INTO @MeetingID,
                          @MeetingPrice;
                      END

                    CLOSE meetingcursor;

                    DEALLOCATE meetingcursor;
                END

              FETCH next FROM activitycursor INTO @ActivityID,
@TypeOfActivity;
          END

      CLOSE activitycursor;

      DEALLOCATE activitycursor;

      COMMIT TRANSACTION;
  END
```

ModifyOrder

K.B.

Cel: Modyfikacja zamówienia na studia.

Parametry:

- `@OrderID INT`: ID zamówienia.

- `@NewUserID INT = NULL`: Nowe ID studenta (opcjonalne).
- `@PaymentDeferred BIT = NULL`: Status odroczonej płatności (opcjonalne).
- `@DeferredDate DATETIME = NULL`: Data odroczenia płatności (opcjonalne).
- `@PaymentLink VARCHAR(255) = NULL`: Link do płatności (opcjonalne).

Walidacje:

1. Sprawdzenie, czy zamówienie o podanym `OrderID` istnieje w tabeli `orders`.
2. Jeśli nie, zgłoszenie błędu i przerwanie transakcji.

Jeśli wszystkie walidacje są pozytywne, modyfikacja zamówienia.

```
CREATE PROCEDURE Modifyorder @OrderID         INT,
                             @NewUserID       INT = NULL,
                             @PaymentDeferred BIT = NULL,
                             @DeferredDate    DATETIME = NULL,
                             @PaymentLink     VARCHAR(255) = NULL
AS
  BEGIN
      BEGIN TRANSACTION;

      UPDATE orders
      SET    studentid = Isnull(@NewUserID, studentid),
             paymentdeferred = Isnull(@PaymentDeferred, paymentdeferred),
             deferreddate = Isnull(@DeferredDate, deferreddate),
             paymentlink = Isnull(@PaymentLink, paymentlink)
      WHERE  orderid = @OrderID;

      IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR ('Zamówienie o podanym ID nie istnieje.',16,1);

            ROLLBACK TRANSACTION;

            RETURN;
        END

      COMMIT TRANSACTION;
  END;
```

DeleteOrder

K.B.

Cel: Usunięcie zamówienia.

Parametry:

- `@OrderID INT`: ID zamówienia.

Walidacje:

1. Sprawdzenie, czy zamówienie o podanym `OrderID` istnieje w tabeli `orders`.
2. Usunięcie szczegółów zamówienia z tabeli `orderdetails` i samego zamówienia z tabeli `orders`.

Jeśli wszystkie walidacje są pozytywne, usunięcie zamówienia.

```
CREATE PROCEDURE Deleteorder @OrderID INT
AS
  BEGIN
      BEGIN TRANSACTION;

      DELETE FROM orderdetails
      WHERE  orderid = @OrderID;

      DELETE FROM orders
      WHERE  orderid = @OrderID;

      IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR ('Zamówienie o podanym ID nie istnieje.',16,1);

            ROLLBACK TRANSACTION;

            RETURN;
        END

      COMMIT TRANSACTION;
  END;
```

ModifyOrderDetail

K.B.

Cel: Modyfikacja szczegółów zamówienia.

Parametry:

- `@DetailID INT`: ID szczegółu zamówienia.
- `@NewActivityID INT = NULL`: Nowe ID aktywności (opcjonalne).
- `@NewTypeOfActivity INT = NULL`: Nowy typ aktywności (opcjonalne).
- `@NewPrice DECIMAL(18,2) = NULL`: Nowa cena (opcjonalne).
- `@NewPaymentStatus BIT = NULL`: Nowy status płatności (opcjonalne).

Walidacje:

1. Sprawdzenie, czy szczegóły zamówienia o podanym `DetailID` istnieją w tabeli `orderdetails`.
2. Sprawdzenie, czy szczegóły zamówienia zostały opłacone. Jeśli tak, modyfikacja jest zabroniona.

Jeśli wszystkie walidacje są pozytywne, modyfikacja szczegółów zamówienia.

```
CREATE PROCEDURE Modifyorderdetail @DetailID          INT,
                                   @NewActivityID     INT = NULL,
                                   @NewTypeOfActivity INT = NULL,
                                   @NewPrice          DECIMAL(18, 2) =
NULL,
                                   @NewPaymentStatus  BIT = NULL
AS
  BEGIN
      BEGIN TRANSACTION;

      DECLARE @PaidStatus BIT;

      SELECT @PaidStatus = paymentstatus
      FROM   orderdetails
      WHERE  detailid = @DetailID;

      IF @PaidStatus = 1
        BEGIN
            RAISERROR (
'Nie można zmodyfikować aktywności, która została opłacona. Usuń ją i
dodaj nową.'
,16,1);

    ROLLBACK TRANSACTION;

    RETURN;
END

    UPDATE orderdetails
    SET    activityid = Isnull(@NewActivityID, activityid),
           typeofactivity = Isnull(@NewTypeOfActivity, typeofactivity),
           price = Isnull(@NewPrice, price),
           paymentstatus = Isnull(@NewPaymentStatus, paymentstatus)
    WHERE  detailid = @DetailID;

    IF @@ROWCOUNT = 0
      BEGIN
          RAISERROR ('Szczegóły zamówienia o podanym ID nie
```

```
istnieją.',16,1)
          ;

          ROLLBACK TRANSACTION;

          RETURN;
      END

    COMMIT TRANSACTION;
END;
```

DeleteOrderDetail

K.B.

Cel: Usunięcie szczegółów zamówienia.

Parametry:

- `@DetailID INT`: ID szczegółu zamówienia.

Walidacje:

1. Sprawdzenie, czy szczegóły zamówienia o podanym `DetailID` istnieją w tabeli `orderdetails`.

Jeśli wszystkie walidacje są pozytywne, usunięcie szczegółów zamówienia.

```
CREATE PROCEDURE Deleteorderdetail @DetailID INT
AS
  BEGIN
      BEGIN TRANSACTION;

      DELETE FROM orderdetails
      WHERE  detailid = @DetailID;

      IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR ('Szczegóły zamówienia o podanym ID nie
istnieją.',16,
                       1)
            ;

            ROLLBACK TRANSACTION;

            RETURN;
        END
```

```
      COMMIT TRANSACTION;
  END;
```

ModifyUserRole

K.B.

Cel: Modyfikacja roli użytkownika.

Parametry:

- `@UserID INT`: ID użytkownika.
- `@NewRoleID INT`: Nowe ID roli.

Walidacje:

1. Sprawdzenie, czy użytkownik ma przypisaną rolę.
2. Jeśli rola istnieje, jej modyfikacja; jeśli nie, dodanie nowej roli.

```
CREATE PROCEDURE Modifyuserrole @UserID    INT,
                                @NewRoleID INT
AS
  BEGIN
      SET nocount ON;

      IF EXISTS (SELECT 1
                 FROM   usersroles
                 WHERE  userid = @UserID)
        BEGIN
            UPDATE usersroles
            SET    roleid = @NewRoleID
            WHERE  userid = @UserID;
        END
      ELSE
        BEGIN
            INSERT INTO usersroles
                        (userid,
                         roleid)
            VALUES      (@UserID,
                         @NewRoleID);
        END
  END;

go
```

DeletedUserRole

K.B.

Cel: Usunięcie roli użytkownika.

Parametry:

- `@UserID INT`: ID użytkownika.

Walidacje:

1. Sprawdzenie, czy użytkownik ma przypisaną rolę.
2. Jeśli rola istnieje, jej usunięcie.

```
CREATE PROCEDURE Deleteuserrole @UserID INT
AS
  BEGIN
      SET nocount ON;

      DELETE FROM usersroles
      WHERE  userid = @UserID;
  END;
```

go

ModifyStudiesResult

K.B.

Cel: Modyfikacja wyniku z przedmiotu dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@StudiesID INT`: ID przedmiotu.
- `@GradeID INT`: ID oceny.

Walidacje:

1. Sprawdzenie, czy wynik z przedmiotu dla danego studenta już istnieje w tabeli `studiesresults`.
2. Jeśli wynik istnieje, modyfikacja oceny; jeśli nie, dodanie nowego wyniku.

```
CREATE PROCEDURE Modifystudiesresult @StudentID INT,
                                     @StudiesID INT,
                                     @GradeID   INT
AS
```

```
BEGIN
    SET nocount ON;

    IF EXISTS (SELECT 1
               FROM   studiesresults
               WHERE  studentid = @StudentID
                      AND studiesid = @StudiesID)
      BEGIN
          UPDATE studiesresults
          SET    gradeid = @GradeID
          WHERE  studentid = @StudentID
                 AND studiesid = @StudiesID;
      END
    ELSE
      BEGIN
          INSERT INTO studiesresults
                      (studentid,
                       studiesid,
                       gradeid)
          VALUES      (@StudentID,
                       @StudiesID,
                       @GradeID);
      END
END;
```

DeleteStudiesResult

K.B.

Cel: Usunięcie wyniku z przedmiotu dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@StudiesID INT`: ID przedmiotu.

Walidacje:

1. Sprawdzenie, czy wynik z przedmiotu dla danego studenta istnieje w tabeli `studiesresults`.

Jeśli walidacja jest pozytywna, usunięcie wyniku.

```
CREATE PROCEDURE Deletestudiesresult @StudentID INT,
                                     @StudiesID INT
AS
```

```
  BEGIN
      SET nocount ON;

      DELETE FROM studiesresults
      WHERE  studentid = @StudentID
             AND studiesid = @StudiesID;
  END;
```

Modifysubjectresult

K.B.

Cel: Modyfikacja wyniku z przedmiotu dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@SubjectID INT`: ID przedmiotu.
- `@GradeID INT`: ID oceny.

Walidacje:

1. Sprawdzenie, czy wynik z przedmiotu dla danego studenta już istnieje w tabeli `subjectsresults`.
2. Jeśli wynik istnieje, modyfikacja oceny; jeśli nie, dodanie nowego wyniku.

```
CREATE PROCEDURE Modifysubjectresult @StudentID INT,
                                     @SubjectID INT,
                                     @GradeID   INT
AS
  BEGIN
      SET nocount ON;

      IF EXISTS (SELECT 1
                 FROM   subjectsresults
                 WHERE  studentid = @StudentID
                        AND subjectid = @SubjectID)
        BEGIN
            UPDATE subjectsresults
            SET    gradeid = @GradeID
            WHERE  studentid = @StudentID
                   AND subjectid = @SubjectID;
        END
      ELSE
        BEGIN
            INSERT INTO subjectsresults
```

```
                        (studentid,
                         subjectid,
                         gradeid)
            VALUES      (@StudentID,
                         @SubjectID,
                         @GradeID);
        END
  END;
```

DeletesubjectResult

K.B.

Cel: Usunięcie wyniku z przedmiotu dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@SubjectID INT`: ID przedmiotu.

Walidacje:

1. Sprawdzenie, czy wynik z przedmiotu dla danego studenta istnieje w tabeli `subjectsresults`.

Jeśli walidacja jest pozytywna, usunięcie wyniku.

```
CREATE PROCEDURE Deletesubjectresult @StudentID INT,
                                     @SubjectID INT
AS
  BEGIN
      SET nocount ON;

      DELETE FROM subjectsresults
      WHERE  studentid = @StudentID
             AND subjectid = @SubjectID;
  END;
```

ModifyInternshipResult

K.B.

Cel: Modyfikacja wyniku z praktyki dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@InternshipID INT`: ID praktyki.
- `@Passed BIT`: Status zdał/nie zdał.

Walidacje:

1. Sprawdzenie, czy wynik z praktyki dla danego studenta już istnieje w tabeli `internshippassed`.
2. Jeśli wynik istnieje, modyfikacja statusu; jeśli nie, dodanie nowego wyniku.

```
CREATE PROCEDURE Modifyinternshipresult @StudentID    INT,
                                        @InternshipID INT,
                                        @Passed       BIT
AS
  BEGIN
      SET nocount ON;

      IF EXISTS (SELECT 1
                 FROM   internshippassed
                 WHERE  studentid = @StudentID
                        AND internshipid = @InternshipID)
        BEGIN
            UPDATE internshippassed
            SET    passed = @Passed
            WHERE  studentid = @StudentID
                   AND internshipid = @InternshipID;
        END
      ELSE
        BEGIN
            INSERT INTO internshippassed
                        (studentid,
                         internshipid,
                         passed)
            VALUES      (@StudentID,
                         @InternshipID,
                         @Passed);
        END
  END;
```

DeleteInternshipResult

K.B.

Cel: Usunięcie wyniku z praktyki dla studenta.

Parametry:

- `@StudentID INT`: ID studenta.
- `@InternshipID INT`: ID praktyki.

Walidacje:

1. Sprawdzenie, czy wynik z praktyki dla danego studenta istnieje w tabeli `internshippassed`.

Jeśli walidacja jest pozytywna, usunięcie wyniku.

```
CREATE PROCEDURE Deleteinternshipresult @StudentID    INT,
                                        @InternshipID INT
AS
  BEGIN
      SET nocount ON;

      DELETE FROM internshippassed
      WHERE  studentid = @StudentID
             AND internshipid = @InternshipID;
  END;
```

ModifyStudyMeetingPresence

`K.B.`

Cel: Modyfikacja obecności studenta na spotkaniu.

Parametry:

- `@StudentID INT`: ID studenta.
- `@MeetingID INT`: ID spotkania.
- `@Present BIT`: Status obecności (1 = obecny, 0 = nieobecny).

Walidacje:

1. Sprawdzenie, czy obecność studenta na spotkaniu istnieje w tabeli `studymeetingpresence`.
2. Jeśli obecność istnieje, modyfikacja statusu obecności; jeśli nie, dodanie nowej obecności.

```
CREATE PROCEDURE Modifystudymeetingpresence @StudentID INT,
                                            @MeetingID INT,
                                            @Present   BIT
AS
  BEGIN
      SET nocount ON;
```

```
    IF EXISTS (SELECT 1
               FROM   studymeetingpresence
               WHERE  studentid = @StudentID
                      AND studymeetingid = @MeetingID)
      BEGIN
          UPDATE studymeetingpresence
          SET    presence = @Present
          WHERE  studentid = @StudentID
                 AND studymeetingid = @MeetingID;
      END
    ELSE
      BEGIN
          INSERT INTO studymeetingpresence
                      (studentid,
                       studymeetingid,
                       presence)
          VALUES      (@StudentID,
                       @MeetingID,
                       @Present);
      END
END;
```

DeleteStudyMeetingPresence

K.B.

Cel: Usunięcie obecności studenta na spotkaniu.

Parametry:

- `@StudentID INT`: ID studenta.
- `@MeetingID INT`: ID spotkania.

Walidacje:

1. Sprawdzenie, czy obecność studenta na spotkaniu istnieje w tabeli `studymeetingpresence`.

Jeśli walidacja jest pozytywna, usunięcie obecności.

```
CREATE PROCEDURE Deletestudymeetingpresence @StudentID INT,
                                            @MeetingID INT
AS
  BEGIN
      SET nocount ON;
```

```
    DELETE FROM studymeetingpresence
    WHERE  studentid = @StudentID
           AND studymeetingid = @MeetingID;
END;
```

ModifyactivityInsteadOfAbsence

K.B.

Cel: Modyfikacja aktywności w zamian za nieobecność na spotkaniu.

Parametry:

- `@StudentID INT`: ID studenta.
- `@AbsenceMeetingID INT`: ID spotkania, na którym student był nieobecny.
- `@ReplacementActivityID INT`: ID aktywności zastępującej nieobecność.
- `@TypeOfActivity INT`: Typ aktywności.

Walidacje:

1. Sprawdzenie, czy typ aktywności jest prawidłowy.
2. Sprawdzenie, czy dla tego studenta i spotkania już istnieje aktywność zastępująca nieobecność.
3. Jeśli aktywność istnieje, jej modyfikacja; jeśli nie, dodanie nowej aktywności.

```
CREATE PROCEDURE Modifyactivityinsteadofabsence @StudentID
INT,
                                                @AbsenceMeetingID
INT,
                                                @ReplacementActivityID
INT,
                                                @TypeOfActivity        INT
AS
  BEGIN
      SET nocount ON;

      IF NOT EXISTS (SELECT 1
                     FROM   activitiestypes
                     WHERE  activitytypeid = @TypeOfActivity)
        BEGIN
            RAISERROR('Nieprawidłowy typ aktywności.',16,1);

            RETURN;
        END

      IF EXISTS (SELECT 1
```

```
                FROM   activityinsteadofabsence
                WHERE  studentid = @StudentID
                       AND meetingid = @AbsenceMeetingID)
      BEGIN
          UPDATE activityinsteadofabsence
          SET    activityid = @ReplacementActivityID
          WHERE  studentid = @StudentID
                 AND meetingid = @AbsenceMeetingID;
      END
    ELSE
      BEGIN
          INSERT INTO activityinsteadofabsence
                      (studentid,
                       meetingid,
                       activityid,
                       typeofactivity)
          VALUES      (@StudentID,
                       @AbsenceMeetingID,
                       @ReplacementActivityID,
                       @TypeOfActivity);
      END
END;
```

Activatedeactivateuser

`J.K.`

Cel: Aktywacja lub deaktywacja konta użytkownika.

Parametry:

- `@UserId INT`: ID użytkownika.
- `@Activate BIT`: Status aktywacji (1 = aktywacja, 0 = deaktywacja).

Walidacje:

1. Sprawdzenie, czy użytkownik istnieje w tabeli `users`.
2. Jeśli użytkownik istnieje, sprawdzenie jego obecnego statusu i zmiana na przeciwny (aktywacja lub deaktywacja).
3. Jeśli użytkownik nie istnieje, zgłoszenie błędu.

Jeśli walidacja jest pozytywna, zmiana statusu konta użytkownika.

```
CREATE PROCEDURE Activatedeactivateuser (@UserId   INT,
                                         @Activate BIT)
AS
  BEGIN
      IF EXISTS (SELECT 1
```

```
                FROM   users
                WHERE  userid = @UserId)
        BEGIN
            DECLARE @CurrentStatus BIT;

            SELECT @CurrentStatus = active
            FROM   users
            WHERE  userid = @UserId;

            IF @CurrentStatus <> @Activate
              BEGIN
                  UPDATE users
                  SET    active = @Activate
                  WHERE  userid = @UserId;

                  IF @Activate = 1
                    BEGIN
                        PRINT 'User account activated.';
                    END
                  ELSE
                    BEGIN
                        PRINT 'User account deactivated.';
                    END
              END
            ELSE
              BEGIN
                  IF @Activate = 1
                    THROW 50002, 'User account is already active.', 1;
                  ELSE
                    THROW 50003, 'User account is already deactivated.',
1;
              END
        END
      ELSE
        BEGIN
            THROW 50001, 'User does not exist.', 1;
        END
  END;
```

FUNKCJE:

CheckIfStudentPassed

O.B.

OPIS
Sprawdza, czy student zdał wszystkie przedmioty związane z podanym kierunkiem studiów.

Funkcja:
Zwraca wartość BIT (0 - student nie zdał, 1 - student zdał).

Argumenty:

- `@StudentID INT` - ID studenta.
- `@StudiesID INT` - ID kierunku studiów.

```
CREATE FUNCTION CheckIfStudentPassed
(
	@StudentID INT,
	@StudiesID INT
)
RETURNS BIT
AS
BEGIN
	DECLARE @Result BIT;


	IF EXISTS (
	SELECT 1
	FROM Subjects s
	LEFT JOIN SubjectsResults sr ON s.SubjectID = sr.SubjectID AND
sr.StudentID = @StudentID
	WHERE s.StudiesID = @StudiesID
		AND (sr.GradeID IS NULL OR sr.GradeID < 2)
	)
	BEGIN
	SET @Result = 0;
	END
	ELSE
	BEGIN
	SET @Result = 1;
	END;
```

```
	RETURN @Result;
END;
GO
```

CheckIfStudentPassedCourse

O.B.

OPIS
Sprawdza, czy student zdał kurs, uwzględniając wymagania dotyczące zaliczonych modułów.

Funkcja:
Zwraca wartość BIT (0 - student nie zdał kursu, 1 - student zdał kurs).

Argumenty:

- `@StudentID INT` - ID studenta.
- `@CourseID INT` - ID kursu.

```
CREATE FUNCTION CheckIfStudentPassedCourse
(
	@StudentID INT,
	@CourseID INT
)
RETURNS BIT
AS
BEGIN
	DECLARE @Result BIT;


	IF NOT EXISTS (
	SELECT 1
	FROM Courses
	INNER JOIN CourseModules ON Courses.CourseID =
CourseModules.CourseID
	LEFT JOIN StationaryCourseMeeting ON CourseModules.ModuleID =
StationaryCourseMeeting.ModuleID
	LEFT JOIN OnlineCourseMeeting ON CourseModules.ModuleID =
OnlineCourseMeeting.ModuleID
	WHERE Courses.CourseID = @CourseID
	AND GETDATE() > OnlineCourseMeeting.EndDate OR GETDATE() >
StationaryCourseMeeting.EndDate
	)
	BEGIN
```

```
        SET @Result = 0;
        RETURN @Result;
        END;


        DECLARE @TotalModules INT;
        SELECT @TotalModules = COUNT(ModuleID)
        FROM VW_CourseModulesPassed
        WHERE CourseID = @CourseID;


        DECLARE @PassedModules INT;
        SELECT @PassedModules = COUNT(ModuleID)
        FROM VW_CourseModulesPassed
        WHERE CourseID = @CourseID
        AND StudentID = @StudentID
        AND Passed = 1;


        IF (@TotalModules > 0 AND @PassedModules >= 0.8 * @TotalModules)
        BEGIN
        SET @Result = 1;
        END
        ELSE
        BEGIN
        SET @Result = 0;
        END;

        RETURN @Result;
END;
GO
```

GetStudentOrders

J.K.

OPIS
Zwraca informacje o zamówieniach studenta, w tym szczegóły zamówień.

Funkcja:
Tabela zawierająca kolumny takie jak `OrderID`, `OrderDate`, `DetailID`, `ActivityID`, `DetailPrice`, `DetailPaymentStatus`.

Argumenty:

- `@StudentID INT` - ID studenta.

```
CREATE FUNCTION GetStudentOrders(@StudentID INT)
RETURNS TABLE
AS
RETURN
(
	SELECT
	o.OrderID,
	o.OrderDate,
	o.DeferredDate,
	od.DetailID,
	od.ActivityID,
	od.Price AS DetailPrice,
	od.PaymentStatus AS DetailPaymentStatus
	FROM Orders o
	JOIN OrderDetails od ON o.OrderID = od.OrderID
	WHERE o.StudentID = @StudentID
);
```

GetProductsFromOrder

J.K.

OPIS
Zwraca szczegóły produktów powiązanych z konkretnym zamówieniem.

Funkcja:
Tabela zawierająca kolumny `OrderID`, `ActivityID`, `TypeOfActivity`.

Argumenty:

- `@OrderID INT` - ID zamówienia.

```
CREATE FUNCTION GetProductsFromOrder
(
	@OrderID INT
)
RETURNS TABLE
AS
RETURN
(
	SELECT
	OD.OrderID,
	OD.ActivityID,
	OD.TypeOfActivity
```

```
	FROM
	OrderDetails OD
	WHERE
	OD.OrderID = @OrderID
);
```

CheckStudentPresenceOnActivity

K.B.

OPIS
Sprawdza obecność studenta na spotkaniu i informuje, czy odrobił nieobecność inną aktywnością.

Funkcja:
Zwraca komunikat tekstowy informujący o obecności lub braku obecności na spotkaniu.

Argumenty:

- `@StudentID INT` - ID studenta.
- `@MeetingID INT` - ID spotkania.

```
CREATE FUNCTION CheckStudentPresenceOnActivity(
	@StudentID INT,
	@MeetingID INT
)
RETURNS NVARCHAR(255)
AS
BEGIN
	DECLARE @PresenceMessage NVARCHAR(255);

	-- Sprawdzanie obecności na spotkaniu
	IF EXISTS (SELECT 1 FROM StudyMeetingPresence
		WHERE StudentID = @StudentID
		AND StudyMeetingID = @MeetingID
		AND Presence = 1)
	BEGIN
	SET @PresenceMessage = 'Student był obecny na tym spotkaniu.';
	END
	ELSE
	BEGIN
	-- Sprawdzanie, czy student odrobił spotkanie inną aktywnością
	IF EXISTS (SELECT 1 FROM ActivityInsteadOfAbsence
			WHERE StudentID = @StudentID
			AND MeetingID = @MeetingID)
	BEGIN
```

```
            SET @PresenceMessage = 'Student odrobił to spotkanie inną
aktywnością.';
      END
      ELSE
      BEGIN
            SET @PresenceMessage = 'Student nie był obecny na tym
spotkaniu ani nie odrobił go inną aktywnością.';
      END
      END

      -- Zwracamy odpowiednią wiadomość
      RETURN @PresenceMessage;
END;
GO
```

GetRemainingSeats

K.B.

OPIS

Oblicza liczbę wolnych miejsc na wybraną aktywność, uwzględniając jej typ.

Funkcja:

Procedura zwraca wartość wyjściową `@RemainingSeats INT`.

Argumenty:

- `@ActivityID INT` - ID aktywności.
- `@TypeOfActivity INT` - Typ aktywności.
- `@RemainingSeats INT OUTPUT` - Liczba wolnych miejsc.

```
CREATE PROCEDURE GetRemainingSeats
      @ActivityID INT,
      @TypeOfActivity INT,
      @RemainingSeats INT OUTPUT
AS
BEGIN
      SET NOCOUNT ON;

      IF @TypeOfActivity = 1
      BEGIN
      SET @RemainingSeats = 2147483647;
      END

      IF @TypeOfActivity = 2
```

```
	BEGIN
	SELECT @RemainingSeats = c.StudentLimit -
		(SELECT ISNULL(COUNT(*), 0)
		FROM OrderDetails od
		WHERE od.ActivityID = c.CourseID AND od.TypeOfActivity = 2)
	FROM Courses c
	WHERE c.CourseID = @ActivityID;

	IF @RemainingSeats IS NULL OR @RemainingSeats < 0
		SET @RemainingSeats = 2147483647;
	END

	IF @TypeOfActivity = 3
	BEGIN
	SELECT @RemainingSeats = s.StudentLimit -
		(SELECT ISNULL(COUNT(*), 0)
		FROM Subjects ss
		INNER JOIN StudyMeetings SM ON SM.SubjectID = ss.SubjectID
		INNER JOIN StudyMeetingPayment SMP ON SMP.MeetingID =
SM.MeetingID
		WHERE ss.StudiesID = s.StudiesID)
	FROM Studies s
	WHERE s.StudiesID = @ActivityID;

	IF @RemainingSeats IS NULL OR @RemainingSeats < 0
		SET @RemainingSeats = 2147483647;
	END

	IF @TypeOfActivity = 4
	BEGIN
	SELECT @RemainingSeats = sm.StudentLimit -
		(SELECT ISNULL(COUNT(*), 0)
		FROM StudyMeetingPayment smp
		WHERE smp.MeetingID = sm.MeetingID)
	FROM StationaryMeetings sm
	INNER JOIN StudyMeetings smt ON smt.MeetingID = sm.MeetingID
	WHERE sm.MeetingID = @ActivityID;

	IF @RemainingSeats IS NULL OR @RemainingSeats < 0
		SET @RemainingSeats = 2147483647;
	END

	-- Jeśli @RemainingSeats nie zostało ustawione
	IF @RemainingSeats IS NULL
	SET @RemainingSeats = -1;

	-- Zwrócenie wyniku
```

```
	SELECT @RemainingSeats AS RemainingSeats;
END;
GO
```

GetAvailableRooms

J.K.

OPIS

Zwraca listę dostępnych sal w danym mieście i przedziale czasowym.

Funkcja:

Tabela zawierająca informacje o dostępnych salach (`RoomID`, `RoomName`, `Street`, `CityID`, `Limit`).

Argumenty:

- `@StartTime DATETIME` - Czas rozpoczęcia.
- `@EndTime DATETIME` - Czas zakończenia.
- `@CityID INT` - ID miasta.

```
CREATE FUNCTION GetAvailableRooms
(
	@StartTime DATETIME,
	@EndTime DATETIME,
	@CityID INT
)
RETURNS TABLE
AS
RETURN
(
	-- Pobranie wolnych sal w podanym czasie w danym mieście
	SELECT Rooms.RoomID, Rooms.RoomName, Rooms.Street, Rooms.CityID,
Rooms.Limit
	FROM Rooms
	WHERE Rooms.CityID = @CityID
	AND EXISTS (
		-- Sprawdzenie, czy CityID istnieje w tabeli Cities
		SELECT 1
		FROM Cities
		WHERE CityID = @CityID
	)
	AND Rooms.RoomID NOT IN
	(
		-- Pobranie wszystkich zajętych sal w danym przedziale
```

```
czasowym
            SELECT RoomID
            FROM VW_RoomsAvailability
            WHERE (@StartTime BETWEEN StartDate AND EndDate)
            OR (@EndTime BETWEEN StartDate AND EndDate)
            OR (StartDate BETWEEN @StartTime AND @EndTime)
            OR (EndDate BETWEEN @StartTime AND @EndTime)
      )
);
```

GetStudentAttendanceAtSubjects

K.B.

OPIS
Zwraca dane dotyczące obecności studenta na przedmiotach.

Funkcja:
Tabela zawierająca informacje o obecności studenta na przedmiotach.

Argumenty:

- `@StudentID INT` - ID studenta.

```
CREATE FUNCTION GetStudentAttendanceAtSubjects (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
      SELECT *
      FROM vw_StudentsAttendanceAtSubjects
      WHERE StudentID = @StudentID
      AND EXISTS (
            SELECT 1
            FROM UsersRoles
            WHERE UserID = @StudentID AND RoleID = 1
      )
);
```

GetStudentResultsFromStudies

O.B.

OPIS
Zwraca wyniki studenta w ramach przedmiotów na kierunkach studiów.

Funkcja:
Tabela zawierająca dane o wynikach studenta (`UserID`, `StudentFirstName`, `StudentLastName`, `SubjectName`, `GradeName`).

Argumenty:

- `@StudentID INT` - ID studenta.

```
CREATE FUNCTION GetStudentResultsFromStudies (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
	SELECT
	U.UserID,
	U.FirstName AS StudentFirstName,
	U.LastName AS StudentLastName,
	S.SubjectName,
	G.GradeName
	FROM SubjectsResults SR
	JOIN Users U ON SR.StudentID = U.UserID
	JOIN Subjects S ON SR.SubjectID = S.SubjectID
	JOIN Grades G ON SR.GradeID = G.GradeID
	WHERE SR.StudentID = @StudentID
	AND EXISTS (
		SELECT 1
		FROM UsersRoles UR
		WHERE UR.UserID = @StudentID AND UR.RoleID = 1
	)
);
```

GetCourseModulesPassed

O.B.

OPIS
Zwraca informacje o modułach kursów zaliczonych przez studenta.

Funkcja:
Tabela z danymi o modułach (`CourseID`, `CourseName`, `ModuleID`, `ModuleName`, `Passed`).

Argumenty:

- `@StudentID INT` - ID studenta.

```sql
CREATE FUNCTION GetCourseModulesPassed (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
	SELECT
	CMP.CourseID,
	CMP.CourseName,
	CMP.ModuleID,
	CMP.ModuleName,
	CMP.FirstName,
	CMP.LastName,
	CMP.Passed
	FROM VW_CourseModulesPassed CMP
	WHERE CMP.StudentID = @StudentID
	AND CMP.Passed = 1
	AND EXISTS (
		SELECT 1
		FROM UsersRoles
		WHERE UserID = @StudentID AND RoleID = 1
	)
);
```

GetFutureMeetingsForStudent

J.K.

OPIS
Zwraca przyszłe spotkania zaplanowane dla studenta.

Funkcja:
Tabela zawierająca szczegóły przyszłych spotkań.

Argumenty:

- `@StudentID INT` - ID studenta.

```sql
CREATE FUNCTION GetFutureMeetingsForStudent (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
	SELECT *
	FROM VW_allUsersFutureMeetings
```

```
	WHERE UserID = @StudentID
	AND EXISTS (
		SELECT 1
		FROM UsersRoles
		WHERE UserID = @StudentID AND RoleID = 1
	)
);
```

GetCurrentMeetingsForStudent

J.K.

OPIS
Zwraca obecne spotkania związane ze studentem.

Funkcja:
Tabela zawierająca informacje o bieżących spotkaniach.

Argumenty:

- `@StudentID INT` - ID studenta.

```
CREATE FUNCTION GetCurrentMeetingsForStudent (@StudentID INT)
RETURNS TABLE
AS
RETURN
(
	SELECT *
	FROM VW_allUsersCurrentMeetings
	WHERE UserID = @StudentID
	AND EXISTS (
		SELECT 1
		FROM UsersRoles
		WHERE UserID = @StudentID AND RoleID = 1
	)
);
```

GetNumberOfHoursOfWorkForAllEmployees

O.B.

OPIS
Oblicza liczbę godzin pracy dla wszystkich pracowników w podanym przedziale czasowym.

Funkcja:
Tabela zawierająca kolumny `EmployeeID`, `liczbagodzinpracy`.

Argumenty:

- `@StartDate DATETIME` - Data początkowa.
- `@EndDate DATETIME` - Data końcowa.

```
CREATE FUNCTION dbo.GetNumberOfHoursOfWorkForAllEmployees
(
	@StartDate datetime,
	@EndDate datetime
)
RETURNS TABLE
	AS
	RETURN
	(
	WITH t1 AS (
	SELECT
		EmployeeID,
		(
			ISNULL(CASE
					WHEN sm.StartTime >= @StartDate AND
sm.EndTime <= @EndDate
					THEN DATEDIFF(minute, sm.EndTime,
sm.StartTime)
					ELSE 0
					END, 0) +
			ISNULL(CASE
					WHEN sm1.StartTime >= @StartDate AND
sm1.EndTime <= @EndDate
					THEN DATEDIFF(minute, sm1.EndTime,
sm1.StartTime)
					ELSE 0
					END, 0) +
			ISNULL(CASE
					WHEN w.StartDate >= @StartDate AND w.EndDate
<= @EndDate
					THEN DATEDIFF(minute, w.EndDate,
w.StartDate)
					ELSE 0
					END, 0) +
			ISNULL(CASE
					WHEN w1.StartDate >= @StartDate AND
w1.EndDate <= @EndDate
					THEN DATEDIFF(minute, w1.EndDate,
w1.StartDate)
					ELSE 0
```

```
                              END, 0) +
                  ISNULL(CASE
                              WHEN ocm.StartDate >= @StartDate AND
ocm.EndDate <= @EndDate
                              THEN DATEDIFF(minute, ocm.EndDate,
ocm.StartDate)
                              ELSE 0
                              END, 0) +
                  ISNULL(CASE
                              WHEN ocm1.StartDate >= @StartDate AND
ocm1.EndDate <= @EndDate
                              THEN DATEDIFF(minute, ocm1.EndDate,
ocm1.StartDate)
                              ELSE 0
                              END, 0) +
                  ISNULL(CASE
                              WHEN scm.StartDate >= @StartDate AND
scm.EndDate <= @EndDate
                              THEN DATEDIFF(minute, scm.EndDate,
scm.StartDate)
                              ELSE 0
                              END, 0) +
                  ISNULL(CASE
                              WHEN scm1.StartDate >= @StartDate AND
scm1.EndDate <= @EndDate
                              THEN DATEDIFF(minute, scm1.EndDate,
scm1.StartDate)
                              ELSE 0
                              END, 0)
                  ) * (-1) as liczbaminut
      FROM Employees
      LEFT OUTER JOIN StudyMeetings AS sm ON sm.LecturerID =
Employees.EmployeeID
      LEFT OUTER JOIN StudyMeetings AS sm1 ON sm1.TranslatorID =
Employees.EmployeeID
      LEFT OUTER JOIN Webinars AS w ON w.TeacherID = Employees.EmployeeID
      LEFT OUTER JOIN Webinars AS w1 ON w1.TranslatorID =
Employees.EmployeeID
      LEFT OUTER JOIN CourseModules AS cm ON cm.LecturerID =
Employees.EmployeeID
      LEFT OUTER JOIN CourseModules AS cm1 ON cm1.TranslatorID =
Employees.EmployeeID
      LEFT OUTER JOIN OnlineCourseMeeting AS ocm ON ocm.ModuleID =
cm.ModuleID
      LEFT OUTER JOIN OnlineCourseMeeting AS ocm1 ON ocm1.ModuleID =
cm1.ModuleID
      LEFT OUTER JOIN StationaryCourseMeeting AS scm ON scm.ModuleID =
```

```
cm.ModuleID
	LEFT OUTER JOIN StationaryCourseMeeting AS scm1 ON scm1.ModuleID =
cm1.ModuleID

	UNION

	SELECT
	internshipCoordinatorID,
	CASE
		WHEN i.StartDate >= @StartDate AND i.EndDate <= @EndDate
			THEN 3000
		ELSE 0
		END
	FROM internship i
)
SELECT
	EmployeeID,
	ROUND(SUM(liczbaminut)/60.0, 2) as liczbagodzinpracy
FROM t1
GROUP BY EmployeeID
)
```

GetUserDiplomasAndCertificates

K.B.

OPIS

Zwraca informacje o dyplomach i certyfikatach użytkownika na podstawie imienia i nazwiska.

Funkcja:

Tabela zawierająca kolumny `CertificateType`, `StudyName`, `GradeName`, `StudyStart`, `StudyEnd`.

Argumenty:

- `@FirstName NVARCHAR(100)` - Imię użytkownika.
- `@LastName NVARCHAR(100)` - Nazwisko użytkownika.

```
CREATE FUNCTION dbo.GetUserDiplomasAndCertificates
(
	@FirstName NVARCHAR(100),
	@LastName NVARCHAR(100)
)
RETURNS TABLE
AS
```

```
RETURN
(
	SELECT
	CASE
		WHEN EXISTS (SELECT 1 FROM Users WHERE FirstName = @FirstName
AND LastName = @LastName) THEN 'Diploma'
		ELSE 'No User Found'
	END AS CertificateType,
	vw_StudentsDiplomas.StudyName,  -- Zakładając, że ta kolumna
istnieje w widoku
	vw_StudentsDiplomas.GradeName,  -- Zakładając, że ta kolumna
istnieje w widoku
	vw_StudentsDiplomas.StudyStart,
	vw_StudentsDiplomas.StudyEnd
	FROM
	vw_StudentsDiplomas
	WHERE
	vw_StudentsDiplomas.FirstName = @FirstName
	AND vw_StudentsDiplomas.LastName = @LastName

	UNION

	SELECT
	'Course Certificate' AS CertificateType,
	vw_CoursesCertificates.CourseName,  -- Zakładając, że ta kolumna
istnieje w widoku
	NULL AS GradeName,
	vw_CoursesCertificates.CourseStart,
	vw_CoursesCertificates.CourseEnd
	FROM
	Users
	INNER JOIN
	vw_CoursesCertificates
	ON
	vw_CoursesCertificates.FirstName = @FirstName
	AND vw_CoursesCertificates.LastName = @LastName
);
```

GetMeetingsInCity

J.K.

OPIS

Zwraca informacje o spotkaniach stacjonarnych w danym mieście.

Funkcja:
Tabela z danymi o spotkaniach (`UserID`, `FirstName`, `LastName`, `ActivityType`, `ActivityName`, `StartTime`, `EndTime`, `RoomName`, `Street`, `PostalCode`, `CityName`).

Argumenty:

- `@cityName NVARCHAR(100)` - Nazwa miasta.

```
CREATE FUNCTION dbo.GetMeetingsInCity
(
	@cityName NVARCHAR(100)
)
RETURNS TABLE
AS
RETURN
(
	SELECT UserID,
		FirstName,
		LastName,
		ActivityType,
		ActivityName,
		StartTime,
		EndTime,
		RoomName,
		Street,
		PostalCode,
		CityName
	FROM VW_allUsersStationaryMeetingsWithRoomAndAddresses
	WHERE CityName = @cityName
);
```

GetTranslatorsMeetings

K.B.

OPIS
Zwraca informacje o wszystkich aktywnościach danego tłumacza.

Funkcja:
Tabela z danymi o spotkaniach (`ActivityType`, `ActivityName`, `SubActivityName`, `StartTime`, `EndTime`).

Argumenty:

- `@FirstName NVARCHAR(100) - imię`

- @LastName NVARCHAR(100) - nazwisko

```
CREATE FUNCTION dbo.GetTranslatorsMeetings
(
    @FirstName NVARCHAR(100),
    @LastName NVARCHAR(100)
)
RETURNS TABLE
AS
RETURN
(

SELECT 'Studia - spotkanie stacjonarne' as ActivityType, ST.StudyName as
ActivityName, S.SubjectName as SubActivityName, StartTime, EndTime
FROM StudyMeetings SM
INNER JOIN StationaryMeetings SMM ON SMM.MeetingID=SM.MeetingID
INNER JOIN Subjects S ON S.SubjectID=SM.SubjectID
INNER JOIN Studies ST ON ST.StudiesID=S.StudiesID
INNER JOIN Users U ON U.UserID = SM.TranslatorID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION

SELECT 'Studia - spotkanie online' as ActivityType, ST.StudyName as
ActivityName, S.SubjectName as SubActivityName, StartTime, EndTime
FROM StudyMeetings SM
INNER JOIN OnlineMeetings OM ON OM.MeetingID=SM.MeetingID
INNER JOIN Subjects S ON S.SubjectID=SM.SubjectID
INNER JOIN Studies ST ON ST.StudiesID=S.StudiesID
INNER JOIN Users U ON U.UserID = SM.TranslatorID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION

SELECT 'Kurs - spotkanie stacjonarne' as ActivityType, c.CourseName as
ActivityName, cm.ModuleName as SubActivityName, StartDate as StartTime,
EndDate as EndTime
FROM StationaryCourseMeeting scm
INNER JOIN CourseModules cm ON cm.ModuleID=scm.ModuleID
INNER JOIN Courses c ON c.CourseID = cm.CourseID
INNER JOIN Users U ON U.UserID = CM.TranslatorID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION
```

```
SELECT 'Kurs - spotkanie online' as ActivityType, c.CourseName as
ActivityName, cm.ModuleName as SubActivityName, StartDate as StartTime,
EndDate as EndTime
FROM OnlineCourseMeeting ocm
INNER JOIN CourseModules cm ON cm.ModuleID=ocm.ModuleID
INNER JOIN Courses c ON c.CourseID = cm.CourseID
INNER JOIN Users U ON U.UserID = CM.TranslatorID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION
SELECT 'Webinar' as ActivityType, webinarName as ActivityName, NULL as
SubActivityName, StartDate as StartTime, EndDate as EndTime
FROM Webinars W
INNER JOIN Users U ON U.UserID = W.TranslatorID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName
);
```

GetLecturersMeetings

K.B.

OPIS

Zwraca informacje o wszystkich aktywnościach danego prowadzącego.

Funkcja:

Tabela z danymi o spotkaniach (`ActivityType`, `ActivityName`, `SubActivityName`, `StartTime`, `EndTime`).

Argumenty:

- `@FirstName NVARCHAR(100) - imię`
- `@LastName NVARCHAR(100) - nazwisko`

```
CREATE FUNCTION dbo.GetLecturersMeetings
(
    @FirstName NVARCHAR(100),
    @LastName NVARCHAR(100)
)
RETURNS TABLE
AS
RETURN
(

SELECT 'Studia - spotkanie stacjonarne' as ActivityType, ST.StudyName as
```

```
ActivityName, S.SubjectName as SubActivityName, StartTime, EndTime
FROM StudyMeetings SM
INNER JOIN StationaryMeetings SMM ON SMM.MeetingID=SM.MeetingID
INNER JOIN Subjects S ON S.SubjectID=SM.SubjectID
INNER JOIN Studies ST ON ST.StudiesID=S.StudiesID
INNER JOIN Users U ON U.UserID = SM.LecturerID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION

SELECT 'Studia - spotkanie online' as ActivityType, ST.StudyName as
ActivityName, S.SubjectName as SubActivityName, StartTime, EndTime
FROM StudyMeetings SM
INNER JOIN OnlineMeetings OM ON OM.MeetingID=SM.MeetingID
INNER JOIN Subjects S ON S.SubjectID=SM.SubjectID
INNER JOIN Studies ST ON ST.StudiesID=S.StudiesID
INNER JOIN Users U ON U.UserID = SM.LecturerID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION

SELECT 'Kurs - spotkanie stacjonarne' as ActivityType, c.CourseName as
ActivityName, cm.ModuleName as SubActivityName, StartDate as StartTime,
EndDate as EndTime
FROM StationaryCourseMeeting scm
INNER JOIN CourseModules cm ON cm.ModuleID=scm.ModuleID
INNER JOIN Courses c ON c.CourseID = cm.CourseID
INNER JOIN Users U ON U.UserID = CM.LecturerID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION

SELECT 'Kurs - spotkanie online' as ActivityType, c.CourseName as
ActivityName, cm.ModuleName as SubActivityName, StartDate as StartTime,
EndDate as EndTime
FROM OnlineCourseMeeting ocm
INNER JOIN CourseModules cm ON cm.ModuleID=ocm.ModuleID
INNER JOIN Courses c ON c.CourseID = cm.CourseID
INNER JOIN Users U ON U.UserID = CM.LecturerID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName

UNION
SELECT 'Webinar' as ActivityType, webinarName as ActivityName, NULL as
SubActivityName, StartDate as StartTime, EndDate as EndTime
FROM Webinars W
INNER JOIN Users U ON U.UserID = W.TeacherID
WHERE U.FirstName = @FirstName AND U.LastName = @LastName
);
```

GetPracticesForCoordinator

O.B.

Funkcja:
Funkcja GetPracticesForCoordinator zwraca tabelę zawierającą szczegóły dotyczące praktyk przypisanych do konkretnego koordynatora, który jest zidentyfikowany na podstawie jego unikalnego ID (parametr @CoordinatorID). Funkcja ta jest zapisana jako **table-valued function** (TVF), co oznacza, że jej wynik stanowi tabelę, którą można wykorzystać w zapytaniach SQL, np. w instrukcji SELECT.

Zawiera:

1. **InternshipID** – ID praktyki (identyfikator praktyki).
2. **StartDate** – Data rozpoczęcia praktyki.
3. **EndDate** – Data zakończenia praktyki.
4. **CoordinatorName** – Imię i nazwisko koordynatora przypisanego do danej praktyki (łączy dane z tabeli Users i Employees).

```
CREATE FUNCTION dbo.fn_GetPracticesForCoordinator
(
	@CoordinatorID INT -- Parametr: ID koordynatora
)
	RETURNS TABLE
	AS
	RETURN
	(
	-- Zapytanie do pobrania praktyk dla koordynatora
	SELECT
		p.InternshipID AS PracticeID,		-- ID praktyki
		p.StartDate AS StartDate,		-- Data rozpoczęcia
		p.EndDate AS EndDate,			-- Data zakończenia
		u.FirstName + ' ' + u.LastName AS CoordinatorName -- Imię i
nazwisko koordynatora
	FROM
		Internship p
			JOIN Employees c ON p.InternshipCoordinatorID =
c.EmployeeID
			JOIN Users u ON c.EmployeeID = u.UserID
	WHERE
		c.EmployeeID = @CoordinatorID -- Filtrujemy po ID koordynatora
```

```
);
```

TRIGGERY:

trg_UpdatePaymentStatus

`J.K.`

Cel: Aktualizacja statusu płatności zamówienia na "udana" i ustawienie ceny na 0, jeśli cena jest pusta lub wynosi 0.

- **Działanie**:
  - Wywoływany po dodaniu lub aktualizacji danych w tabeli `OrderDetails`.
  - Sprawdza, czy cena (`Price`) jest `NULL` lub równa `0`.
  - Aktualizuje kolumny `PaymentStatus` na `'udana'` i `Price` na `0`.

```
CREATE TRIGGER trg_UpdatePaymentStatus
ON OrderDetails
AFTER INSERT, UPDATE
AS
BEGIN
      UPDATE od
      SET od.PaymentStatus = 'udana',
      od.Price = 0
      FROM OrderDetails od
      INNER JOIN inserted i ON od.DetailID = i.DetailID
      WHERE i.Price IS NULL OR i.Price = 0;
END;
```

---

trg_SetPaymentDeferred

`J.K.`

Cel: Ustawia status odroczenia płatności w zamówieniach.

- **Działanie**:
  - Wywoływany po dodaniu lub aktualizacji danych w tabeli `Orders`.
  - Jeśli pole `DeferredDate` wstawionego wiersza nie jest puste, ustawia `PaymentDeferred` na `1`, w przeciwnym razie na `0`.

```
CREATE TRIGGER trg_SetPaymentDeferred
ON Orders
AFTER INSERT, UPDATE
AS
BEGIN
	SET NOCOUNT ON;
	UPDATE o
	SET o.PaymentDeferred = CASE
	WHEN i.DeferredDate IS NOT NULL THEN 1
	ELSE 0
	END
	FROM Orders o
	INNER JOIN inserted i ON o.OrderID = i.OrderID;
END;
```

BeforeOrderDetailsInsert

K.B.

Cel: Walidacja przed wstawieniem nowych szczegółów zamówienia w tabeli `OrderDetails`.

- **Działanie**:
  - Sprawdza, czy użytkownik jest aktywny.
  - Weryfikuje, czy liczba uczestników na kursach, studiach lub spotkaniach nie przekracza limitów.
  - Wywołuje błąd i cofa transakcję, jeśli limity są przekroczone lub użytkownik jest nieaktywny.

```
CREATE TRIGGER BeforeOrderDetailsInsert
ON OrderDetails
FOR INSERT
AS
BEGIN
BEGIN TRANSACTION
	DECLARE @ActivityID INT;
	DECLARE @TypeOfActivity INT;
	DECLARE @Remaining INT;


	IF @TypeOfActivity = 4
	BEGIN
	SET @RemainingSeats = dbo.GetRemainingSeatsFn(@ActivityID,
@TypeOfActivity);
```

```
	IF (@RemainingSeats<=0)
	BEGIN
		RAISERROR ('Brak miejsc na spotkaniu o ID %d.', 16, 1,
@ActivityID);		ROLLBACK TRANSACTION;
		RETURN;
	END
	END


	IF @TypeOfActivity = 3
	BEGIN
	SET @RemainingSeats = dbo.GetRemainingSeatsFn(@ActivityID,
@TypeOfActivity);

	IF (@RemainingSeats<=0)
	BEGIN
		RAISERROR ('Brak miejsc na studiach o ID %d.', 16, 1,
@ActivityID);
		ROLLBACK TRANSACTION;
		RETURN;
	END
	END

	IF @TypeOfActivity = 2
	BEGIN
	SET @RemainingSeats = dbo.GetRemainingSeatsFn(@ActivityID,
@TypeOfActivity);

	IF (@RemainingSeats<=0)
	BEGIN
		RAISERROR ('Brak miejsc na kursie  o ID %d.', 16, 1,
@ActivityID);

		ROLLBACK TRANSACTION;
		RETURN;
	END
	END
END;
```

BeforeOrderDetailsUpdate

K.B.

Cel: Walidacja podczas aktualizacji szczegółów zamówienia.

- **Działanie**:
  - Sprawdza, czy użytkownik jest aktywny.

- Jeśli zmienia się przypisana aktywność (`ActivityID`), weryfikuje limity uczestników dla nowej aktywności.
- Wywołuje błąd i cofa transakcję w przypadku przekroczenia limitu lub dezaktywowanego użytkownika.

```
CREATE TRIGGER BeforeOrderDetailsUpdate
ON OrderDetails
FOR UPDATE
AS
BEGIN
	DECLARE @UserID INT;
	DECLARE @OldActivityID INT;
	DECLARE @NewActivityID INT;
	DECLARE @TypeOfActivity INT;
	DECLARE @StudentLimit INT;
	DECLARE @TotalBooked INT;

	SELECT @UserID = O.StudentID,
		@OldActivityID = OD.ActivityID,
		@NewActivityID = I.ActivityID,
		@TypeOfActivity = OD.TypeOfActivity
	FROM INSERTED I
	INNER JOIN DELETED D ON I.DetailID = D.DetailID
	INNER JOIN Orders O ON O.OrderID = I.OrderID
	INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
	WHERE I.OrderID = D.OrderID;


	IF NOT EXISTS (SELECT 1 FROM Users WHERE UserID = @UserID AND Active
= 1)
	BEGIN
	RAISERROR ('Użytkownik o ID %d jest nieaktywny i nie może
zaktualizować zamówienia.', 16, 1, @UserID);
	ROLLBACK TRANSACTION;
	RETURN;
	END

	IF @OldActivityID != @NewActivityID
	BEGIN
	IF @TypeOfActivity = 4
	BEGIN
		SELECT @StudentLimit = SM.StudentLimit
		FROM StationaryMeetings SM
		INNER JOIN StudyMeetings SM2 ON SM.MeetingID = SM2.MeetingID
		WHERE SM2.MeetingID = @NewActivityID;
```

```
            SELECT @TotalBooked = COUNT(O.OrderID)
            FROM Orders O
            INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
            WHERE OD.ActivityID = @NewActivityID
            AND OD.TypeOfActivity = @TypeOfActivity;

            IF (@TotalBooked + 1) > @StudentLimit
            BEGIN
                  RAISERROR ('Przekroczono limit miejsc na spotkaniu
stacjonarnym dla aktywności o ID %d.', 16, 1, @NewActivityID);
                  ROLLBACK TRANSACTION;
                  RETURN;
            END
      END

      IF @TypeOfActivity = 3
      BEGIN
            SELECT @StudentLimit = S.StudentLimit
            FROM Studies S WHERE S.StudiesID = @NewActivityID;

            SELECT @TotalBooked = COUNT(OD.OrderID)
            FROM OrderDetails OD
            WHERE OD.ActivityID = @NewActivityID
            AND OD.TypeOfActivity = @TypeOfActivity;

            IF (@TotalBooked + 1) > @StudentLimit
            BEGIN
                  RAISERROR ('Przekroczono limit miejsc na studiach dla
aktywności o ID %d.', 16, 1, @NewActivityID);
                  ROLLBACK TRANSACTION;
                  RETURN;
            END
      END

      IF @TypeOfActivity = 2
      BEGIN
            SELECT @StudentLimit = C.StudentLimit
            FROM Courses C WHERE C.CourseID = @NewActivityID;

            SELECT @TotalBooked = COUNT(OD.OrderID)
            FROM OrderDetails OD
            WHERE OD.ActivityID = @NewActivityID
            AND OD.TypeOfActivity = @TypeOfActivity;

            IF (@TotalBooked + 1) > @StudentLimit
            BEGIN
```

```
                  RAISERROR ('Przekroczono limit miejsc na kursie dla
aktywności o ID %d.', 16, 1, @NewActivityID);
                  ROLLBACK TRANSACTION;
                  RETURN;
            END
      END
      END
END;
```

PreventStudyUpdateAfterStart

K.B.

Cel: Blokuje aktualizację danych studiów po rozpoczęciu pierwszego spotkania.

- **Działanie**:
  - Sprawdza datę rozpoczęcia najwcześniejszego spotkania powiązanego ze studiami.
  - Wywołuje błąd, jeśli aktualizacja dotyczy studiów już rozpoczętych.

```
CREATE TRIGGER PreventStudyUpdateAfterStart
ON Studies
FOR UPDATE
AS
BEGIN
      DECLARE @StudiesID INT;
      DECLARE @EarliestMeetingDate DATETIME;
      DECLARE @CurrentDate DATETIME = GETDATE();


      SELECT @StudiesID = StudiesID FROM INSERTED;


      SELECT @EarliestMeetingDate = MIN(SM.StartTime)
      FROM StudyMeetings SM
      INNER JOIN Subjects S ON SM.SubjectID = S.SubjectID
      WHERE S.StudiesID = @StudiesID;

      IF @EarliestMeetingDate <= @CurrentDate
      BEGIN
      DECLARE @FormattedDate VARCHAR(20) = CONVERT(VARCHAR(20),
@EarliestMeetingDate, 120);
      RAISERROR ('Nie można edytować danych studiów po rozpoczęciu
studiów. Najwcześniejsze spotkanie miało miejsce %s.', 16, 1,
@FormattedDate);
```

```
	ROLLBACK TRANSACTION;
	RETURN;
	END
END;
```

PreventSubjectUpdateAfterStart

K.B.

Cel: Blokuje aktualizację przedmiotu po rozpoczęciu studiów.

- **Działanie**:
  - Analogiczne do `PreventStudyUpdateAfterStart`, ale działa na poziomie przedmiotu.

```
CREATE TRIGGER PreventSubjectUpdateAfterStart
ON Subjects
FOR UPDATE
AS
BEGIN
	DECLARE @StudiesID INT;
	DECLARE @EarliestMeetingDate DATETIME;
	DECLARE @CurrentDate DATETIME = GETDATE();

	SELECT @StudiesID = StudiesID FROM INSERTED;

	SELECT @EarliestMeetingDate = MIN(SM.StartTime)
	FROM StudyMeetings SM
	INNER JOIN Subjects S ON SM.SubjectID = S.SubjectID
	WHERE S.StudiesID = @StudiesID;

	IF @EarliestMeetingDate <= @CurrentDate
	BEGIN
	DECLARE @FormattedDate VARCHAR(20) = CONVERT(VARCHAR(20),
@EarliestMeetingDate, 120);
	RAISERROR ('Nie można edytować danych przedmiotu po rozpoczęciu
studiów. Najwcześniejsze spotkanie miało miejsce %s.', 16, 1,
@FormattedDate);
	ROLLBACK TRANSACTION;
	RETURN;
	END
END;
```

CheckStudyMeetingOverlap

O.B.

Cel: Zapobieganie nakładaniu się terminów spotkań w ramach tych samych studiów.

- **Działanie**:
  - Sprawdza, czy nowo wstawione spotkanie nie pokrywa się czasowo z istniejącymi.
  - Jeśli występuje konflikt, wstrzymuje operację i wyświetla komunikat o błędzie.

```
CREATE TRIGGER CheckStudyMeetingOverlap
ON StudyMeetings
INSTEAD OF INSERT
AS
BEGIN
	-- Sprawdzanie pokrycia czasowego
	IF EXISTS (
	SELECT 1
	FROM Inserted i
	INNER JOIN Subjects s ON i.SubjectID = s.SubjectID
	INNER JOIN StudyMeetings sm ON sm.SubjectID = s.SubjectID
	WHERE sm.EndTime > i.StartTime AND sm.StartTime < i.EndTime
		AND sm.MeetingID != i.MeetingID -- Aby nie porównywać tego
samego spotkania
		AND s.StudiesID = (SELECT StudiesID FROM Subjects WHERE
SubjectID = i.SubjectID)
	)
	BEGIN
	RAISERROR ('Spotkanie pokrywa się czasowo z innym spotkaniem w
ramach tych samych studiów.', 16, 1);
	RETURN;
	END

	-- Jeśli nie ma konfliktów czasowych, wstaw wiersz do StudyMeetings
	INSERT INTO StudyMeetings (SubjectID, LecturerID, MeetingType,
MeetingPrice, MeetingPriceForOthers, StartTime, EndTime, LanguageID,
TranslatorID)
	SELECT SubjectID, LecturerID, MeetingType, MeetingPrice,
MeetingPriceForOthers, StartTime, EndTime, LanguageID, TranslatorID
	FROM Inserted;
END;
```

CheckUserRoleInsert

K.B.

Cel: Walidacja przy wstawianiu ról użytkowników.

- **Działanie**:
  - Sprawdza, czy `UserID` istnieje w tabeli `Users`, a `RoleID` w tabeli `Roles`.
  - W przypadku braku odpowiadającego użytkownika lub roli, wywołuje błąd.

```
CREATE TRIGGER CheckUserRoleInsert
ON UsersRoles
INSTEAD OF INSERT
AS
BEGIN
	-- Sprawdzenie, czy istnieje użytkownik w tabeli Users
	IF EXISTS (
	SELECT 1
	FROM Inserted i
	LEFT JOIN Users u ON i.UserID = u.UserID
	WHERE u.UserID IS NULL
	)
	BEGIN
	RAISERROR ('Podany użytkownik nie istnieje w tabeli Users.', 16, 1);
	RETURN;
	END

	-- Sprawdzenie, czy istnieje rola w tabeli Roles
	IF EXISTS (
	SELECT 1
	FROM Inserted i
	LEFT JOIN Roles r ON i.RoleID = r.RoleID
	WHERE r.RoleID IS NULL
	)
	BEGIN
	RAISERROR ('Podana rola nie istnieje w tabeli Roles.', 16, 1);
	RETURN;
	END

	-- Jeśli wszystkie warunki są spełnione, wstaw dane do UsersRoles
	INSERT INTO UsersRoles (UserID, RoleID)
	SELECT UserID, RoleID
	FROM Inserted;
END;
```

CheckEmployeeExistsInUsers

O.B.

Cel: Walidacja wstawiania pracowników w tabeli `Employees`.

- **Działanie**:
  - Weryfikuje, czy `EmployeeID` istnieje w tabeli `Users`.
  - W przypadku braku użytkownika o podanym `EmployeeID`, operacja jest anulowana.

```
CREATE TRIGGER CheckEmployeeExistsInUsers
ON Employees
INSTEAD OF INSERT
AS
BEGIN
      -- Sprawdzenie, czy EmployeeID istnieje w tabeli Users
      IF EXISTS (
      SELECT 1
      FROM Inserted i
      LEFT JOIN Users u ON i.EmployeeID = u.UserID
      WHERE u.UserID IS NULL
      )
      BEGIN
      RAISERROR ('EmployeeID nie istnieje w tabeli Users.', 16, 1);
      RETURN;
      END

      -- Jeśli warunek jest spełniony, wstaw dane do tabeli Employees
      INSERT INTO Employees (EmployeeID, HireDate, DegreeID)
      SELECT EmployeeID, HireDate, DegreeID
      FROM Inserted;
END;
```

CheckTranslatedLanguageValidity

O.B.

Cel: Walidacja przy dodawaniu języków tłumaczeń.

- **Działanie**:
  - Sprawdza istnienie tłumacza (`TranslatorID`) w tabeli `Users` i języka (`LanguageID`) w tabeli `Languages`.
  - Wywołuje błąd, jeśli którykolwiek z warunków nie jest spełniony.

```
CREATE TRIGGER CheckTranslatedLanguageValidity
ON TranslatedLanguage
```

```
INSTEAD OF INSERT
AS
BEGIN
	-- Sprawdzenie, czy TranslatorID istnieje w tabeli Users oraz
LanguageID w tabeli Languages
	IF EXISTS (
	SELECT 1
	FROM Inserted i
	LEFT JOIN Users u ON i.TranslatorID = u.UserID
	LEFT JOIN Languages l ON i.LanguageID = l.LanguageID
	WHERE u.UserID IS NULL OR l.LanguageID IS NULL
	)
	BEGIN
	RAISERROR ('TranslatorID nie istnieje w tabeli Users lub LanguageID
nie istnieje w tabeli Languages.', 16, 1);
	RETURN;
	END

	-- Jeśli warunek jest spełniony, wstaw dane do tabeli
TranslatedLanguage
	INSERT INTO TranslatedLanguage (TranslatorID, LanguageID)
	SELECT TranslatorID, LanguageID
	FROM Inserted;
END;
```

CheckUserDeactivation

O.B.

Cel: Walidacja przy dezaktywacji użytkownika.

- **Działanie**:
  - Sprawdza, czy dezaktywowany użytkownik (student lub pracownik) nie jest przypisany do trwających lub przyszłych wydarzeń.
  - Anuluje operację i wywołuje błąd, jeśli użytkownik jest powiązany z aktywnymi wydarzeniami.

```
CREATE TRIGGER CheckUserDeactivation
ON Users
INSTEAD OF UPDATE
AS
BEGIN
	-- Sprawdzenie, czy użytkownik jest dezaktywowany
	IF EXISTS (
	SELECT 1
```

```
	FROM Inserted i
	JOIN Deleted d ON i.UserID = d.UserID
	WHERE d.Active = 1 AND i.Active = 0
	)
	BEGIN
	-- Sprawdzamy każdą dezaktywację osobno
	DECLARE @UserID INT, @RoleID INT;
	DECLARE UserCursor CURSOR FOR
	SELECT i.UserID, ur.RoleID
	FROM Inserted i
	JOIN Deleted d ON i.UserID = d.UserID
	JOIN UsersRoles ur ON i.UserID = ur.UserID
	WHERE d.Active = 1 AND i.Active = 0;

	OPEN UserCursor;
	FETCH NEXT FROM UserCursor INTO @UserID, @RoleID;

	WHILE @@FETCH_STATUS = 0
	BEGIN
		-- Jeśli użytkownik jest studentem (RoleID = 1)
		IF @RoleID = 1
		BEGIN
			IF EXISTS (
			SELECT 1
			FROM Orders o
			JOIN OrderDetails od ON o.OrderID = od.OrderID
			LEFT JOIN StudyMeetings sm ON od.ActivityID =
sm.MeetingID AND od.TypeOfActivity = 4
			LEFT JOIN Webinars w ON od.ActivityID = w.WebinarID AND
od.TypeOfActivity = 1
			LEFT JOIN Courses c ON od.ActivityID = c.CourseID AND
od.TypeOfActivity = 2
			LEFT JOIN Studies s ON od.ActivityID = s.StudiesID AND
od.TypeOfActivity = 3
			WHERE o.StudentID = @UserID
				AND (
				(sm.StartTime > GETDATE()) OR
				(w.StartDate > GETDATE()) OR
				(EXISTS (
					SELECT 1
					FROM StationaryCourseMeeting scm
					WHERE scm.ModuleID = c.CourseID AND
scm.StartDate > GETDATE()
				)) OR
				(EXISTS (
					SELECT 1
					FROM OnlineCourseMeeting ocm
```

```
                              WHERE ocm.ModuleID = c.CourseID AND
ocm.StartDate > GETDATE()
                        ))
                        )
                  )
                  BEGIN
                  ROLLBACK TRANSACTION;
                  RAISERROR ('Nie można dezaktywować użytkownika, który
jest studentem i jest przypisany do przyszłego wydarzenia.', 16, 1);
                  RETURN;
                  END
            END
            ELSE
            BEGIN
                  -- Jeśli użytkownik jest pracownikiem (RoleID ≠ 1)
                  IF EXISTS (
                  SELECT 1
                  FROM StudyMeetings sm
                  WHERE (sm.LecturerID = @UserID OR sm.TranslatorID =
@UserID)
                        AND sm.StartTime <= GETDATE()
                        AND sm.EndTime >= GETDATE()
                  )
                  OR EXISTS (
                  SELECT 1
                  FROM Courses c
                  WHERE c.CourseCoordinatorID = @UserID
                  )
                  OR EXISTS (
                  SELECT 1
                  FROM Webinars w
                  WHERE (w.TeacherID = @UserID OR w.TranslatorID =
@UserID)
                        AND w.StartDate <= GETDATE()
                        AND w.EndDate >= GETDATE()
                  )
                  OR EXISTS (
                  SELECT 1
                  FROM CourseModules cm
                  LEFT JOIN StationaryCourseMeetings scm ON
StationaryCourseMeetings.ModuleID = cm.ModuleID
                  LEFT JOIN OnlineCourseMeetings ocm ON
OnlineCourseMeetings.ModuleID = cm.ModuleID
                  WHERE (cm.LecturerID = @UserID OR cm.TranslatorID =
@UserID)
                        AND ((scm.StartTime <= GETDATE()
                        AND scm.EndTime >= GETDATE()) OR ocm.StartTime
```

```
<=GETDATE() AND ocm.EndTime >= GETDATE())
                  )
                  BEGIN
                  ROLLBACK TRANSACTION;
                  RAISERROR ('Nie można dezaktywować użytkownika, który
jest pracownikiem i jest przypisany do obecnie trwającego wydarzenia jako
prowadzący, koordynator lub tłumacz.', 16, 1);
                  RETURN;
                  END
            END

            FETCH NEXT FROM UserCursor INTO @UserID, @RoleID;
      END;

      CLOSE UserCursor;
      DEALLOCATE UserCursor;
      END;

      -- Jeśli wszystkie warunki są spełnione, wykonaj aktualizację
      UPDATE Users
      SET Active = i.Active
      FROM Users u
      JOIN Inserted i ON u.UserID = i.UserID;
END;
```

INDEXY:

IDX_ActivitiesTypes_TypeName

Index: Zapewnia unikalność wartości w kolumnie `TypeName` w tabeli `ActivitiesTypes`, gwarantując brak duplikatów typów aktywności.

```
CREATE UNIQUE INDEX IDX_ActivitiesTypes_TypeName
ON ActivitiesTypes (TypeName);
```

IDX_Cities_CityName_CountryID

Index: Gwarantuje unikalność kombinacji nazwy miasta (`CityName`) i kraju (`CountryID`) w tabeli `Cities`, zapobiegając dodaniu dwóch miast o tej samej nazwie w tym samym kraju.

```
CREATE UNIQUE INDEX IDX_Cities_CityName_CountryID
ON Cities (CityName, CountryID);
```

IDX_Countries_CountryName

Index: Wymusza unikalność nazw krajów (`CountryName`) w tabeli `Countries`, dzięki czemu każdy kraj ma tylko jedną unikalną nazwę.

```
CREATE UNIQUE INDEX IDX_Countries_CountryName
ON Countries (CountryName);
```

IDX_Degrees_DegreeName

Index: Zapewnia unikalność nazw stopni naukowych (`DegreeName`) w tabeli `Degrees`.

```
CREATE UNIQUE INDEX IDX_Degrees_DegreeName
ON Degrees (DegreeName);
```

IDX_FormOfActivity_TypeName

Index: Wymusza unikalność nazw form aktywności (`TypeName`) w tabeli `FormOfActivity`.

```
CREATE UNIQUE INDEX IDX_FormOfActivity_TypeName
ON FormOfActivity (TypeName);
```

IDX_Grades_GradeName

Index: Gwarantuje unikalność nazw ocen (`GradeName`) w tabeli `Grades`.

```
CREATE UNIQUE INDEX IDX_Grades_GradeName
ON Grades (GradeName);
```

IDX_Languages_LanguageName

Index: Zapewnia unikalność nazw języków (`LanguageName`) w tabeli `Languages`.

```
CREATE UNIQUE INDEX IDX_Languages_LanguageName
ON Languages (LanguageName);
```

IDX_Roles_RoleName

Index: Wymusza unikalność nazw ról (`RoleName`) w tabeli `Roles`.

```
CREATE UNIQUE INDEX IDX_Roles_RoleName
ON Roles (RoleName);
```

IDX_Users_Email

Index: Gwarantuje unikalność adresów e-mail (`Email`) w tabeli `Users`.

```
CREATE UNIQUE INDEX IDX_Users_Email
ON Users (Email);
```

IDX_Users_CityID

Index: Przyspiesza wyszukiwanie rekordów użytkowników według identyfikatora miasta (`CityID`) w tabeli `Users`.

```
CREATE INDEX IDX_Users_CityID
ON Users (CityID);
```

IDX_Employees_DegreeID

Index: Ułatwia szybkie wyszukiwanie pracowników na podstawie przypisanego stopnia naukowego (`DegreeID`) w tabeli `Employees`.

```
CREATE INDEX IDX_Employees_DegreeID
ON Employees (DegreeID);
```

IDX_Employees_EmployeeID

Index: Przyspiesza operacje związane z identyfikatorem pracownika (`EmployeeID`) w tabeli `Employees`.

```
CREATE INDEX IDX_Employees_EmployeeID
ON Employees (EmployeeID);
```

IDX_Courses_CourseCoordinatorID

Index: Usprawnia wyszukiwanie kursów według identyfikatora koordynatora kursu (`CourseCoordinatorID`) w tabeli `Courses`.

```
CREATE INDEX IDX_Courses_CourseCoordinatorID
ON Courses (CourseCoordinatorID);
```

IDX_Courses_CoursePrice

Index: Optymalizuje wyszukiwanie kursów na podstawie ceny kursu (`CoursePrice`) w tabeli `Courses`.

```
CREATE INDEX IDX_Courses_CoursePrice
ON Courses (CoursePrice);
```

IDX_Studies_StudiesCoordinatorID

Index: Przyspiesza wyszukiwanie studiów według identyfikatora koordynatora studiów (`StudiesCoordinatorID`) w tabeli `Studies`.

```
CREATE INDEX IDX_Studies_StudiesCoordinatorID
ON Studies (StudiesCoordinatorID);
```

IDX_Studies_StudyPrice

Index: Ułatwia szybkie wyszukiwanie studiów na podstawie ich ceny (`StudyPrice`) w tabeli `Studies`.

```
CREATE INDEX IDX_Studies_StudyPrice
ON Studies (StudyPrice);
```

IDX_Subjects_TeacherID

Index: Optymalizuje operacje wyszukiwania przedmiotów na podstawie przypisanego nauczyciela (`TeacherID`) w tabeli `Subjects`.

```
CREATE INDEX IDX_Subjects_TeacherID
ON Subjects (TeacherID);
```

IDX_Subjects_StudiesID

Index: Ułatwia szybkie wyszukiwanie przedmiotów według przypisanych studiów (`StudiesID`) w tabeli `Subjects`.

```
CREATE INDEX IDX_Subjects_StudiesID
ON Subjects (StudiesID);
```

IDX_StudyMeetings_LecturerID

Index: Przyspiesza operacje związane z wyszukiwaniem spotkań studiów na podstawie identyfikatora wykładowcy (`LecturerID`) w tabeli `StudyMeetings`.

```
CREATE INDEX IDX_StudyMeetings_LecturerID
ON StudyMeetings (LecturerID);
```

IDX_StudyMeetings_TranslatorID

Index: Ułatwia wyszukiwanie spotkań studiów z tłumaczem (`TranslatorID`) w tabeli `StudyMeetings`.

```
CREATE INDEX IDX_StudyMeetings_TranslatorID
ON StudyMeetings (TranslatorID);
```

IDX_StudyMeetings_SubjectID

Index: Optymalizuje operacje na tabeli `StudyMeetings` dla kolumny przedmiotu (`SubjectID`).

```
CREATE INDEX IDX_StudyMeetings_SubjectID
ON StudyMeetings (SubjectID);
```

IDX_StudyMeetingPresence_StudentID

Index: Usprawnia wyszukiwanie obecności studentów na spotkaniach (`StudentID`) w tabeli `StudyMeetingPresence`.

```
CREATE INDEX IDX_StudyMeetingPresence_StudentID
ON StudyMeetingPresence (StudentID);
```

IDX_StudyMeetingPresence_StudyMeetingID

Index: Przyspiesza operacje na danych obecności w oparciu o spotkanie studiów (`StudyMeetingID`).

```
CREATE INDEX IDX_StudyMeetingPresence_StudyMeetingID
ON StudyMeetingPresence (StudyMeetingID);
```

IDX_CourseModules_CourseID

Index: Optymalizuje wyszukiwanie modułów kursów według identyfikatora kursu (`CourseID`) w tabeli `CourseModules`.

```
CREATE INDEX IDX_CourseModules_CourseID
ON CourseModules (CourseID);
```

IDX_CourseModules_LecturerID

Index: Ułatwia szybkie wyszukiwanie modułów na podstawie wykładowcy (`LecturerID`) w tabeli `CourseModules`.

```
CREATE INDEX IDX_CourseModules_LecturerID
ON CourseModules (LecturerID);
```

IDX_CourseModules_TranslatorID

Index: Usprawnia operacje na tabeli `CourseModules` dla kolumny tłumacza (`TranslatorID`).

```
CREATE INDEX IDX_CourseModules_TranslatorID
ON CourseModules (TranslatorID);
```

IDX_CourseModulesPassed_ModuleID

Index: Ułatwia szybkie wyszukiwanie ukończonych modułów (`ModuleID`) w tabeli `CourseModulesPassed`.

```
CREATE INDEX IDX_CourseModulesPassed_ModuleID
ON CourseModulesPassed (ModuleID);
```

IDX_CourseModulesPassed_StudentID

Index: Przyspiesza operacje związane z wyszukiwaniem ukończonych modułów na podstawie studenta (`StudentID`).

```
CREATE INDEX IDX_CourseModulesPassed_StudentID
ON CourseModulesPassed (StudentID);
```

IDX_Internship_InternshipCoordinatorID

Index: Przyspiesza wyszukiwanie praktyk na podstawie koordynatora praktyk (`InternshipCoordinatorID`) w tabeli `Internship`.

```
CREATE INDEX IDX_Internship_InternshipCoordinatorID
ON Internship (InternshipCoordinatorID);
```

IDX_Internship_StudiesID

Index: Optymalizuje operacje na tabeli `Internship` w oparciu o przypisane studia (`StudiesID`).

```
CREATE INDEX IDX_Internship_StudiesID
ON Internship (StudiesID);
```

IDX_InternshipPassed_InternshipID

Index: Ułatwia wyszukiwanie zaliczonych praktyk na podstawie identyfikatora praktyki (`InternshipID`).

```
CREATE INDEX IDX_InternshipPassed_InternshipID
ON InternshipPassed (InternshipID);
```

IDX_InternshipPassed_StudentID

Index: Optymalizuje operacje wyszukiwania zaliczonych praktyk dla danego studenta (`StudentID`).

```
CREATE INDEX IDX_InternshipPassed_StudentID
ON InternshipPassed (StudentID);
```

IDX_OrderDetails_OrderID

Index: Usprawnia wyszukiwanie szczegółów zamówienia według identyfikatora zamówienia (`OrderID`).

```
CREATE INDEX IDX_OrderDetails_OrderID
ON OrderDetails (OrderID);
```

IDX_OrderDetails_TypeOfActivity

Index: Ułatwia szybkie wyszukiwanie szczegółów zamówienia na podstawie typu aktywności (`TypeOfActivity`).

```
CREATE INDEX IDX_OrderDetails_TypeOfActivity
ON OrderDetails (TypeOfActivity);
```

IDX_Orders_StudentID

Index: Optymalizuje operacje wyszukiwania zamówień w oparciu o identyfikator studenta (`StudentID`).

```
CREATE INDEX IDX_Orders_StudentID
ON Orders (StudentID);
```

IDX_PaymentsAdvances_DetailID

Index: Przyspiesza wyszukiwanie zaliczek na płatności na podstawie szczegółów (`DetailID`) w tabeli `PaymentsAdvances`.

```
CREATE INDEX IDX_PaymentsAdvances_DetailID
ON PaymentsAdvances (DetailID);
```

IDX_Rooms_CityID

Index: Optymalizuje operacje na tabeli `Rooms` dla kolumny miasta (`CityID`).

```
CREATE INDEX IDX_Rooms_CityID
ON Rooms (CityID);
```

IDX_StationaryMeetings_RoomID

Index: Ułatwia szybkie wyszukiwanie spotkań stacjonarnych według sali (`RoomID`).

```
CREATE INDEX IDX_StationaryMeetings_RoomID
ON StationaryMeetings (RoomID);
```

IDX_StationaryMeetings_MeetingID

Index: Przyspiesza operacje na tabeli `StationaryMeetings` w oparciu o identyfikator spotkania (`MeetingID`).

```
CREATE INDEX IDX_StationaryMeetings_MeetingID
ON StationaryMeetings (MeetingID);
```

IDX_StationaryCourseMeeting_ModuleID

Index: Optymalizuje operacje na tabeli `StationaryCourseMeeting` dla kolumny modułu (`ModuleID`).

```
CREATE INDEX IDX_StationaryCourseMeeting_ModuleID
ON StationaryCourseMeeting (ModuleID);
```

IDX_StationaryCourseMeeting_RoomID

Index: Ułatwia szybkie wyszukiwanie spotkań stacjonarnych kursu według sali (`RoomID`).

```
CREATE INDEX IDX_StationaryCourseMeeting_RoomID
ON StationaryCourseMeeting (RoomID);
```

IDX_Webinars_TeacherID

Index: Przyspiesza operacje wyszukiwania webinarów na podstawie nauczyciela (`TeacherID`).

```
CREATE INDEX IDX_Webinars_TeacherID
ON Webinars (TeacherID);
```

IDX_Webinars_TranslatorID

Index: Ułatwia szybkie wyszukiwanie webinarów z tłumaczem (`TranslatorID`).

```
CREATE INDEX IDX_Webinars_TranslatorID
ON Webinars (TranslatorID);
```

IDX_Webinars_LanguageID

Index: Optymalizuje operacje na tabeli `Webinars` w oparciu o język webinaru (`LanguageID`).

```
CREATE INDEX IDX_Webinars_LanguageID
ON Webinars (LanguageID);
```

IDX_UsersRoles_UserID

Index: Przyspiesza wyszukiwanie ról przypisanych do użytkowników na podstawie identyfikatora użytkownika (`UserID`).

```
CREATE INDEX IDX_UsersRoles_UserID
ON UsersRoles (UserID);
```

IDX_UsersRoles_RoleID

Index: Optymalizuje operacje związane z wyszukiwaniem ról według ich identyfikatora (`RoleID`).

```
CREATE INDEX IDX_UsersRoles_RoleID
ON UsersRoles (RoleID);
```

ROLE:

1. Rola: `admin`

Rola o najwyższych uprawnieniach związanych z tworzeniem kopii zapasowych systemu. Może wykonywać następujące operacje:

- **Kopie zapasowe baz danych**: Uprawnienia do wykonywania pełnych kopii zapasowych bazy danych (`BACKUP DATABASE`).
- **Kopie zapasowe logów transakcyjnych**: Uprawnienia do tworzenia kopii zapasowych logów transakcyjnych (`BACKUP LOG`).

```
CREATE ROLE admin;
GRANT BACKUP DATABASE TO admin;
GRANT BACKUP LOG TO admin;
```

---

2. Rola: `director`

Rola menadżerska z szerokimi uprawnieniami do przeglądania danych oraz wykonywania operacji zarządczych. Kluczowe funkcjonalności:

Uprawnienia odczytu (SELECT):

- Dostęp do raportów, takich jak:
  - `bilocation_report`: Informacje o nakładających się aktywnościach użytkowników.
  - `defferedPayments`: Dane o opóźnionych płatnościach.
- Wgląd w widoki związane z aktywnościami użytkowników (kursy, spotkania, webinaria, itp.).
- Przegląd koordynatorów, prowadzących, tłumaczy oraz szczegółów związanych z kursami, studiami i webinariami.
- Wgląd w dane o dochodach z różnych form edukacji (`VW_IncomeFromCourses`, `VW_IncomeFromStudies`, itp.).
- Dostęp do danych o obecnościach, ocenach, dyplomach oraz demografii studentów (`vw_Students`, `vw_StudentsGradesWithSubjects`).

Uprawnienia wykonawcze (EXECUTE):

- Zarządzanie użytkownikami i ich rolami (`AddUser`, `DeleteUser`, `ModifyUserRole`).
- Tworzenie i edycja kursów, modułów, przedmiotów, webinariów oraz spotkań.
- Zarządzanie danymi o zamówieniach, płatnościach oraz dostępności miejsc na kursach i spotkaniach.
- Sprawdzanie obecności, wyników oraz statusu ukończenia aktywności przez studentów.

```
CREATE ROLE director;
```

```
GRANT SELECT ON bilocation_report TO director;
GRANT SELECT ON defferedPayments TO director;
GRANT SELECT ON VW_allOrderedActivities TO director;
GRANT SELECT ON VW_allPastUsersWebinars TO director;
GRANT SELECT ON VW_allUsersCourseMeetings TO director;
GRANT SELECT ON VW_allUsersCurrentCourseMeetings TO director;
GRANT SELECT ON VW_allUsersCurrentMeetings TO director;
GRANT SELECT ON VW_allUsersCurrentStudyMeetings TO director;
GRANT SELECT ON VW_allUsersCurrentWebinars TO director;
GRANT SELECT ON VW_allUsersFutureCourseMeetings TO director;
GRANT SELECT ON VW_allUsersFutureMeetings TO director;
GRANT SELECT ON VW_allUsersFutureStudyMeetings TO director;
GRANT SELECT ON VW_allUsersFutureWebinars TO director;
GRANT SELECT ON VW_allUsersMeetings TO director;
GRANT SELECT ON VW_allUsersPastCourseMeetings TO director;
GRANT SELECT ON VW_allUsersPastMeetings TO director;
GRANT SELECT ON VW_allUsersPastStudyMeetings TO director;
GRANT SELECT ON VW_allUsersStationaryMeetingsWithRoomAndAddresses TO
director;
GRANT SELECT ON VW_allUsersStudyMeetings TO director;
GRANT SELECT ON VW_allUsersWebinars TO director;
GRANT SELECT ON VW_BilocationBetweenAllActivities TO director;
GRANT SELECT ON vw_CourseCoordinators TO director;
GRANT SELECT ON vw_CourseModulesLanguages TO director;
GRANT SELECT ON VW_CourseModulesPassed TO director;
GRANT SELECT ON vw_CoursesCertificates TO director;
GRANT SELECT ON vw_CoursesCertificatesAddresses TO director;
GRANT SELECT ON VW_CoursesCoordinators TO director;
GRANT SELECT ON vw_CoursesLecturers TO director;
GRANT SELECT ON VW_CoursesStartDateEndDate TO director;
GRANT SELECT ON vw_CoursesWithModules TO director;
GRANT SELECT ON VW_CurrentCoursesPassed TO director;
GRANT SELECT ON vw_Debtors TO director;
GRANT SELECT ON vw_EmployeeDegrees TO director;
GRANT SELECT ON VW_EmployeesFunctionsAndSeniority TO director;
GRANT SELECT ON VW_IncomeFromAllFormOfEducation TO director;
GRANT SELECT ON VW_IncomeFromCourses TO director;
GRANT SELECT ON VW_IncomeFromStudies TO director;
GRANT SELECT ON VW_IncomeFromWebinars TO director;
GRANT SELECT ON vw_InternshipCoordinators TO director;
GRANT SELECT ON vw_InternshipDetails TO director;
GRANT SELECT ON vw_InternshipsParticipants TO director;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnCourses TO director;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnStudies TO director;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnWebinars TO director;
GRANT SELECT ON vw_LecturerMeetings TO director;
GRANT SELECT ON vw_Lecturers TO director;
```

```
GRANT SELECT ON vw_MeetingsWithAbsences TO director;
GRANT SELECT ON vw_NumberOfHoursOfWOrkForAllEmployees TO director;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureAllFormOfEducation TO
director;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureCourses TO director;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureStudyMeetings TO
director;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureWebinars TO director;
GRANT SELECT ON vw_OrdersWithDetails TO director;
GRANT SELECT ON vw_PresenceOnPastStudyMeeting TO director;
GRANT SELECT ON VW_RoomsAvailability TO director;
GRANT SELECT ON vw_Students TO director;
GRANT SELECT ON vw_StudentsAttendanceAtSubjects TO director;
GRANT SELECT ON vw_StudentsDiplomas TO director;
GRANT SELECT ON vw_StudentsGradesWithSubjects TO director;
GRANT SELECT ON VW_StudentsPlaceOfLive TO director;
GRANT SELECT ON VW_StudiesCoordinators TO director;
GRANT SELECT ON VW_StudiesStartDateEndDate TO director;
GRANT SELECT ON vw_StudyCoordinator TO director;
GRANT SELECT ON VW_StudyMeetingDurationTimeNumberOfStudents TO director;
GRANT SELECT ON vw_StudyMeetings TO director;
GRANT SELECT ON VW_StudyMeetingsPresenceWithFirstNameLastNameDate TO
director;
GRANT SELECT ON vw_SubjectsEachGradesNumber TO director;
GRANT SELECT ON vw_TranslatorsWithLanguages TO director;
GRANT SELECT ON VW_UsersDiplomasAddresses TO director;
GRANT SELECT ON VW_UsersPersonalData TO director;
GRANT SELECT ON vw_UsersWithRoles TO director;
GRANT SELECT ON vw_WebinarsLanguages TO director;
GRANT SELECT ON vw_WebinarsWithDetails TO director;
GRANT SELECT ON vw_WebinarTeachers TO director;

GRANT EXECUTE ON ActivateDeactivateUser TO director;
GRANT EXECUTE ON AddCity TO director;
GRANT EXECUTE ON AddCountry TO director;
GRANT EXECUTE ON AddCourseMeeting TO director;
GRANT EXECUTE ON AddCourseModule TO director;
GRANT EXECUTE ON AddEmployee TO director;
GRANT EXECUTE ON AddInternship TO director;
GRANT EXECUTE ON AddLanguage TO director;
GRANT EXECUTE ON AddNewCourse TO director;
GRANT EXECUTE ON AddNewStudy TO director;
GRANT EXECUTE ON AddNewSubject TO director;
GRANT EXECUTE ON AddNewWebinar TO director;
GRANT EXECUTE ON AddOrderWithDetails TO director;
GRANT EXECUTE ON AddRoom TO director;
GRANT EXECUTE ON AddStudyResult TO director;
```

```
GRANT EXECUTE ON AddSubjectMeeting TO director;
GRANT EXECUTE ON AddSubjectResult TO director;
GRANT EXECUTE ON AddTranslatedLanguage TO director;
GRANT EXECUTE ON AddUser TO director;
GRANT EXECUTE ON AddUserRole TO director;
GRANT EXECUTE ON AddUserWithRoleAndEmployee TO director;
GRANT EXECUTE ON DeleteCourse TO director;
GRANT EXECUTE ON DeleteCourseMeeting TO director;
GRANT EXECUTE ON DeleteCourseModule TO director;
GRANT EXECUTE ON DeleteEmployee TO director;
GRANT EXECUTE ON DeleteInternship TO director;
GRANT EXECUTE ON DeleteInternshipResult TO director;
GRANT EXECUTE ON DeleteLanguageFromTranslatedLanguage TO director;
GRANT EXECUTE ON DeleteOrder TO director;
GRANT EXECUTE ON DeleteOrderDetail TO director;
GRANT EXECUTE ON DeleteStudiesResult TO director;
GRANT EXECUTE ON DeleteStudy TO director;
GRANT EXECUTE ON DeleteStudyMeeting TO director;
GRANT EXECUTE ON DeleteSubject TO director;
GRANT EXECUTE ON DeleteSubjectResult TO director;
GRANT EXECUTE ON DeleteTranslatedLanguage TO director;
GRANT EXECUTE ON DeleteUser TO director;
GRANT EXECUTE ON DeleteUserRole TO director;
GRANT EXECUTE ON DeleteWebinar TO director;
GRANT EXECUTE ON GetRemainingSeats TO director;
GRANT EXECUTE ON ModifyInternshipResult TO director;
GRANT EXECUTE ON ModifyOrder TO director;
GRANT EXECUTE ON ModifyOrderDetail TO director;
GRANT EXECUTE ON ModifyStudiesResult TO director;
GRANT EXECUTE ON ModifySubjectResult TO director;
GRANT EXECUTE ON ModifyUserRole TO director;
GRANT EXECUTE ON UpdateCity TO director;
GRANT EXECUTE ON UpdateCountry TO director;
GRANT EXECUTE ON UpdateCourse TO director;
GRANT EXECUTE ON UpdateCourseModule TO director;
GRANT EXECUTE ON UpdateCourseModuleMeeting TO director;
GRANT EXECUTE ON UpdateEmployee TO director;
GRANT EXECUTE ON UpdateInternship TO director;
GRANT EXECUTE ON UpdateLanguage TO director;
GRANT EXECUTE ON UpdateRoom TO director;
GRANT EXECUTE ON UpdateStudies TO director;
GRANT EXECUTE ON UpdateStudyMeeting TO director;
GRANT EXECUTE ON UpdateSubject TO director;
GRANT EXECUTE ON UpdateTranslatedLanguage TO director;
GRANT EXECUTE ON UpdateUser TO director;
GRANT EXECUTE ON UpdateWebinar TO director;
```

```
GRANT EXECUTE ON CheckIfStudentPassed TO director;
GRANT EXECUTE ON CheckIfStudentPassedCourse TO director;
GRANT EXECUTE ON CheckStudentPresenceOnActivity TO director;
GRANT EXECUTE ON fn_diagramobjects TO director;
GRANT SELECT ON GetAvailableRooms TO director;
GRANT SELECT ON GetCourseModulesPassed TO director;
GRANT SELECT ON GetCurrentMeetingsForStudent TO director;
GRANT SELECT ON GetFutureMeetingsForStudent TO director;
GRANT SELECT ON GetMeetingsInCity TO director;
GRANT SELECT ON GetNumberOfHoursOfWorkForAllEmployees TO director;
GRANT SELECT ON GetProductsFromOrder TO director;
GRANT SELECT ON GetStudentOrders TO director;
GRANT SELECT ON GetStudentResultsFromStudies TO director;
GRANT SELECT ON GetUserDiplomasAndCertificates TO director;
```

---

3. Rola: `study_coordinator`

Rola skoncentrowana na zarządzaniu studiami i spotkaniami studyjnymi.

Uprawnienia odczytu (SELECT):

- Wgląd w dane dotyczące:
  - Spotkań studyjnych (przyszłych, przeszłych i bieżących).
  - Studentów oraz ich osiągnięć, obecności i dyplomów.
  - Koordynatorów oraz szczegółów praktyk i tłumaczeń.
- Informacje o dostępności sal (`VW_RoomsAvailability`).

Uprawnienia wykonawcze (EXECUTE):

- Dodawanie wyników studiów i spotkań.
- Aktualizacja szczegółów studiów, spotkań oraz praktyk.
- Zarządzanie obecnościami na spotkaniach.

```
CREATE ROLE study_coordinator;
GRANT SELECT ON VW_allUsersCurrentStudyMeetings TO study_coordinator;
GRANT SELECT ON VW_allUsersFutureStudyMeetings TO study_coordinator;
GRANT SELECT ON VW_allUsersPastStudyMeetings TO study_coordinator;
GRANT SELECT ON vw_InternshipCoordinators TO study_coordinator;
GRANT SELECT ON vw_InternshipDetails TO study_coordinator;
GRANT SELECT ON vw_InternshipsParticipants TO study_coordinator;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnStudies TO study_coordinator;
GRANT SELECT ON vw_LecturerMeetings TO study_coordinator;
GRANT SELECT ON vw_Lecturers TO study_coordinator;
GRANT SELECT ON vw_MeetingsWithAbsences TO study_coordinator;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureStudyMeetings TO
```

```
study_coordinator;
GRANT SELECT ON vw_PresenceOnPastStudyMeeting TO study_coordinator;
GRANT SELECT ON VW_RoomsAvailability TO study_coordinator;
GRANT SELECT ON vw_Students TO study_coordinator;
GRANT SELECT ON vw_StudentsAttendanceAtSubjects TO study_coordinator;
GRANT SELECT ON vw_StudentsDiplomas TO study_coordinator;
GRANT SELECT ON vw_StudentsGradesWithSubjects TO study_coordinator;
GRANT SELECT ON VW_StudentsPlaceOfLive TO study_coordinator;
GRANT SELECT ON VW_StudiesStartDateEndDate TO study_coordinator;
GRANT SELECT ON VW_StudyMeetingDurationTimeNumberOfStudents TO
study_coordinator;
GRANT SELECT ON vw_StudyMeetings TO study_coordinator;
GRANT SELECT ON VW_StudyMeetingsPresenceWithFirstNameLastNameDate TO
study_coordinator;
GRANT SELECT ON vw_SubjectsEachGradesNumber TO study_coordinator;
GRANT SELECT ON vw_TranslatorsWithLanguages TO study_coordinator;
GRANT SELECT ON VW_UsersDiplomasAddresses TO study_coordinator;

GRANT EXECUTE ON AddStudyResult TO study_coordinator;
GRANT EXECUTE ON AddSubjectMeeting TO study_coordinator;
GRANT EXECUTE ON DeleteStudyMeeting TO study_coordinator;
GRANT EXECUTE ON ModifyStudiesResult TO study_coordinator;
GRANT EXECUTE ON UpdateInternship TO study_coordinator;
GRANT EXECUTE ON UpdateStudies TO study_coordinator;
GRANT EXECUTE ON UpdateStudyMeeting TO study_coordinator;
GRANT EXECUTE ON UpdateSubject TO study_coordinator;
GRANT EXECUTE ON AddStudyResult TO study_coordinator;
GRANT EXECUTE ON DeleteStudiesResult TO study_coordinator;

GRANT EXECUTE ON CheckStudentPresenceOnActivity TO study_coordinator;
GRANT SELECT ON GetAvailableRooms TO study_coordinator;
```

---

4. Rola: `course_coordinator`

Rola odpowiedzialna za organizację i zarządzanie kursami.

Uprawnienia odczytu (SELECT):

- Dostęp do szczegółów kursów, takich jak:
  - Moduły, języki, prowadzący, koordynatorzy i uczestnicy.
  - Wyniki ukończonych modułów (`VW_CourseModulesPassed`).
  - Liczba zapisanych studentów na przyszłe kursy.
  - Informacje o certyfikatach i ich adresach (`vw_CoursesCertificatesAddresses`).

Uprawnienia wykonawcze (EXECUTE):

- Aktualizacja szczegółów kursów i modułów.
- Zarządzanie ukończeniem modułów przez uczestników.
- Sprawdzanie dostępności sal i wyników kursów.

```
CREATE ROLE course_coordinator;
GRANT SELECT ON VW_allUsersCourseMeetings TO course_coordinator;
GRANT SELECT ON VW_allUsersCurrentCourseMeetings TO course_coordinator;
GRANT SELECT ON VW_allUsersFutureCourseMeetings TO course_coordinator;
GRANT SELECT ON VW_allUsersPastCourseMeetings TO course_coordinator;
GRANT SELECT ON vw_CourseCoordinators TO course_coordinator;
GRANT SELECT ON vw_CourseModulesLanguages TO course_coordinator;
GRANT SELECT ON VW_CourseModulesPassed TO course_coordinator;
GRANT SELECT ON vw_CoursesCertificates TO course_coordinator;
GRANT SELECT ON vw_CoursesCertificatesAddresses TO course_coordinator;
GRANT SELECT ON VW_CoursesCoordinators TO course_coordinator;
GRANT SELECT ON vw_CoursesLecturers TO course_coordinator;
GRANT SELECT ON VW_CoursesStartDateEndDate TO course_coordinator;
GRANT SELECT ON vw_CoursesWithModules TO course_coordinator;
GRANT SELECT ON VW_CurrentCoursesPassed TO course_coordinator;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnCourses TO course_coordinator;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureCourses TO
course_coordinator;
GRANT SELECT ON vw_TranslatorsWithLanguages TO course_coordinator;

GRANT EXECUTE ON UpdateCourse TO course_coordinator;
GRANT EXECUTE ON UpdateCourseModule TO course_coordinator;
GRANT EXECUTE ON UpdateCourseModuleMeeting TO course_coordinator;
GRANT EXECUTE ON UpdateCourseModulePassed TO course_coordinator;

GRANT SELECT ON GetAvailableRooms TO course_coordinator;
GRANT SELECT ON GetCourseModulesPassed TO course_coordinator;
```

---

5. Rola: `webinars_coordinator`

Specjalistyczna rola do zarządzania webinariami.

Uprawnienia odczytu (SELECT):

- Dostęp do szczegółów dotyczących webinariów:
  - Języki, tłumacze, prowadzący i uczestnicy.
  - Liczba zapisanych studentów na przyszłe webinaria.
  - Informacje o odbytych i bieżących webinariach.

Uprawnienia wykonawcze (EXECUTE):

- Aktualizacja szczegółów dotyczących webinariów.

```
CREATE ROLE webinars_coordinator;
GRANT SELECT ON VW_allPastUsersWebinars TO webinars_coordinator;
GRANT SELECT ON VW_allUsersCurrentWebinars TO webinars_coordinator;
GRANT SELECT ON VW_allUsersFutureWebinars TO webinars_coordinator;
GRANT SELECT ON VW_allUsersWebinars TO webinars_coordinator;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnWebinars TO
webinars_coordinator;
GRANT SELECT ON vw_Lecturers TO webinars_coordinator;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureWebinars TO
webinars_coordinator;
GRANT SELECT ON vw_TranslatorsWithLanguages TO webinars_coordinator;
GRANT SELECT ON vw_WebinarsLanguages TO webinars_coordinator;
GRANT SELECT ON vw_WebinarsWithDetails TO webinars_coordinator;
GRANT SELECT ON vw_WebinarTeachers TO webinars_coordinator;

GRANT EXECUTE ON UpdateWebinar TO webinars_coordinator;
```

---

6. Rola: `accountant`

Rola finansowa, odpowiedzialna za monitorowanie finansów.

Uprawnienia odczytu (SELECT):

- Dostęp do danych finansowych, takich jak:
  - Dochody z kursów, studiów i webinariów.
  - Opóźnione płatności i zadłużenia.
  - Szczegóły zamówień oraz dane o studentach.

Uprawnienia wykonawcze (EXECUTE):

- Monitorowanie godzin pracy pracowników (`GetNumberOfHoursOfWorkForAllEmployees`).

```
CREATE ROLE acountant;
GRANT SELECT ON defferedPayments TO acountant;
GRANT SELECT ON vw_Debtors TO acountant;
GRANT SELECT ON VW_IncomeFromCourses TO acountant;
GRANT SELECT ON VW_IncomeFromStudies TO acountant;
GRANT SELECT ON VW_IncomeFromWebinars TO acountant;
GRANT SELECT ON vw_NumberOfHoursOfWOrkForAllEmployees TO acountant;
GRANT SELECT ON vw_OrdersWithDetails TO acountant;
```

```
GRANT SELECT ON vw_Students TO acountant;

GRANT SELECT ON GetNumberOfHoursOfWorkForAllEmployees TO acountant;
```

---

7. Rola: `secretary`

Rola administracyjna z szerokim zakresem uprawnień do odczytu oraz ograniczonym zakresem uprawnień wykonawczych.

Uprawnienia odczytu (SELECT):

- Dostęp do większości widoków i raportów związanych z aktywnościami, studentami, prowadzącymi oraz kursami.

Uprawnienia wykonawcze (EXECUTE):

- Zarządzanie użytkownikami (`ActivateDeactivateUser`).
- Monitorowanie obecności, wyników oraz statusu studentów.
- Zarządzanie aktywnościami zamiast obecności.

```
CREATE ROLE secretary;
GRANT SELECT ON bilocation_report TO secretary;
GRANT SELECT ON defferedPayments TO secretary;
GRANT SELECT ON VW_allOrderedActivities TO secretary;
GRANT SELECT ON VW_allPastUsersWebinars TO secretary;
GRANT SELECT ON VW_allUsersCourseMeetings TO secretary;
GRANT SELECT ON VW_allUsersCurrentCourseMeetings TO secretary;
GRANT SELECT ON VW_allUsersCurrentMeetings TO secretary;
GRANT SELECT ON VW_allUsersCurrentStudyMeetings TO secretary;
GRANT SELECT ON VW_allUsersCurrentWebinars TO secretary;
GRANT SELECT ON VW_allUsersFutureCourseMeetings TO secretary;
GRANT SELECT ON VW_allUsersFutureMeetings TO secretary;
GRANT SELECT ON VW_allUsersFutureStudyMeetings TO secretary;
GRANT SELECT ON VW_allUsersFutureWebinars TO secretary;
GRANT SELECT ON VW_allUsersMeetings TO secretary;
GRANT SELECT ON VW_allUsersPastCourseMeetings TO secretary;
GRANT SELECT ON VW_allUsersPastMeetings TO secretary;
GRANT SELECT ON VW_allUsersPastStudyMeetings TO secretary;
GRANT SELECT ON VW_allUsersStationaryMeetingsWithRoomAndAddresses TO
secretary;
GRANT SELECT ON VW_allUsersStudyMeetings TO secretary;
GRANT SELECT ON VW_allUsersWebinars TO secretary;
GRANT SELECT ON VW_BilocationBetweenAllActivities TO secretary;
GRANT SELECT ON vw_CourseCoordinators TO secretary;
GRANT SELECT ON vw_CourseModulesLanguages TO secretary;
```

```
GRANT SELECT ON VW_CourseModulesPassed TO secretary;
GRANT SELECT ON vw_CoursesCertificates TO secretary;
GRANT SELECT ON vw_CoursesCertificatesAddresses TO secretary;
GRANT SELECT ON VW_CoursesCoordinators TO secretary;
GRANT SELECT ON vw_CoursesLecturers TO secretary;
GRANT SELECT ON VW_CoursesStartDateEndDate TO secretary;
GRANT SELECT ON vw_CoursesWithModules TO secretary;
GRANT SELECT ON VW_CurrentCoursesPassed TO secretary;
GRANT SELECT ON vw_Debtors TO secretary;
GRANT SELECT ON vw_EmployeeDegrees TO secretary;
GRANT SELECT ON VW_EmployeesFunctionsAndSeniority TO secretary;
GRANT SELECT ON vw_InternshipCoordinators TO secretary;
GRANT SELECT ON vw_InternshipDetails TO secretary;
GRANT SELECT ON vw_InternshipsParticipants TO secretary;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnCourses TO secretary;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnStudies TO secretary;
GRANT SELECT ON VW_LanguagesAndTranslatorsOnWebinars TO secretary;
GRANT SELECT ON vw_LecturerMeetings TO secretary;
GRANT SELECT ON vw_Lecturers TO secretary;
GRANT SELECT ON vw_MeetingsWithAbsences TO secretary;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureAllFormOfEducation TO
secretary;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureCourses TO secretary;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureStudyMeetings TO
secretary;
GRANT SELECT ON VW_NumberOfStudentsSignUpForFutureWebinars TO secretary;
GRANT SELECT ON vw_PresenceOnPastStudyMeeting TO secretary;
GRANT SELECT ON VW_RoomsAvailability TO secretary;
GRANT SELECT ON vw_Students TO secretary;
GRANT SELECT ON vw_StudentsAttendanceAtSubjects TO secretary;
GRANT SELECT ON vw_StudentsDiplomas TO secretary;
GRANT SELECT ON vw_StudentsGradesWithSubjects TO secretary;
GRANT SELECT ON VW_StudentsPlaceOfLive TO secretary;
GRANT SELECT ON VW_StudiesCoordinators TO secretary;
GRANT SELECT ON VW_StudiesStartDateEndDate TO secretary;
GRANT SELECT ON vw_StudyCoordinator TO secretary;
GRANT SELECT ON VW_StudyMeetingDurationTimeNumberOfStudents TO secretary;
GRANT SELECT ON vw_StudyMeetings TO secretary;
GRANT SELECT ON VW_StudyMeetingsPresenceWithFirstNameLastNameDate TO
secretary;
GRANT SELECT ON vw_SubjectsEachGradesNumber TO secretary;
GRANT SELECT ON vw_TranslatorsWithLanguages TO secretary;
GRANT SELECT ON VW_UsersDiplomasAddresses TO secretary;
GRANT SELECT ON VW_UsersPersonalData TO secretary;
GRANT SELECT ON vw_UsersWithRoles TO secretary;
GRANT SELECT ON vw_WebinarsLanguages TO secretary;
GRANT SELECT ON vw_WebinarsWithDetails TO secretary;
```

```
GRANT SELECT ON vw_WebinarTeachers TO secretary;

GRANT EXECUTE ON ActivateDeactivateUser TO secretary;
GRANT EXECUTE ON AddActivityInsteadOfPresence TO secretary;
GRANT EXECUTE ON GetRemainingSeats TO secretary;
GRANT EXECUTE ON ModifyActivityInsteadOfAbsence TO secretary;

GRANT EXECUTE ON CheckIfStudentPassed TO secretary;
GRANT EXECUTE ON CheckIfStudentPassedCourse TO secretary;
GRANT EXECUTE ON CheckStudentPresenceOnActivity TO secretary;
GRANT EXECUTE ON fn_diagramobjects TO secretary;
GRANT SELECT ON GetAvailableRooms TO secretary;
GRANT SELECT ON GetCourseModulesPassed TO secretary;
GRANT SELECT ON GetCurrentMeetingsForStudent TO secretary;
GRANT SELECT ON GetFutureMeetingsForStudent TO secretary;
GRANT SELECT ON GetMeetingsInCity TO secretary;
GRANT SELECT ON GetProductsFromOrder TO secretary;
GRANT SELECT ON GetStudentOrders TO secretary;
GRANT SELECT ON GetStudentResultsFromStudies TO secretary;
GRANT SELECT ON GetUserDiplomasAndCertificates TO secretary;
```

---

8. Rola: `lecturer`

Rola skoncentrowana na interakcji z uczestnikami oraz ocenianiu ich osiągnięć.

Uprawnienia wykonawcze (EXECUTE):

- Dodawanie, edycja i usuwanie obecności na spotkaniach.
- Zarządzanie wynikami studentów z przedmiotów.

```
CREATE ROLE lecturer;
GRANT EXECUTE ON AddStudyMeetingPresence TO lecturer;
GRANT EXECUTE ON DeleteStudyMeetingPresence TO lecturer;
GRANT EXECUTE ON ModifyStudyMeetingPresence TO lecturer;
GRANT EXECUTE ON ModifySubjectResult TO lecturer;
GRANT EXECUTE ON AddSubjectResult TO lecturer;
GRANT EXECUTE ON DeleteSubjectResult TO lecturer;
```

---

9. Rola: `internship_coordinator`

Rola odpowiedzialna za organizację praktyk.

Uprawnienia wykonawcze (EXECUTE):

- Dodawanie, edycja i usuwanie wyników praktyk.

```
CREATE ROLE internship_coordinator;
GRANT EXECUTE ON AddInternshipResult TO internship_coordinator;
GRANT EXECUTE ON DeleteInternshipResult TO internship_coordinator;
GRANT EXECUTE ON ModifyInternshipResult TO internship_coordinator;
```

---

10. Rola: `translator`

Rola bez przypisanych szczególnych uprawnień. Może pełnić dodatkowe zadania w zależności od potrzeb organizacji.

```
CREATE ROLE translator;
```

---

11. Rola: `student`

Rola uczestnika zajęć.

Uprawnienia wykonawcze (EXECUTE):

- Zarządzanie własnymi zamówieniami (`AddOrderWithDetails`, `DeleteOrder`, itp.).
- Sprawdzanie obecności, wyników oraz statusu ukończenia kursów i aktywności.

```
CREATE ROLE student;
GRANT EXECUTE ON AddOrderWithDetails TO student;
GRANT EXECUTE ON DeleteOrder TO student;
GRANT EXECUTE ON DeleteOrderDetail TO student;
GRANT EXECUTE ON ModifyOrderDetail TO student;

GRANT EXECUTE ON CheckIfStudentPassed TO student;
GRANT EXECUTE ON CheckIfStudentPassedCourse TO student;
GRANT EXECUTE ON CheckStudentPresenceOnActivity TO student;
GRANT SELECT ON GetCurrentMeetingsForStudent TO student;
GRANT SELECT ON GetFutureMeetingsForStudent TO student;
GRANT SELECT ON GetProductsFromOrder TO student;
GRANT SELECT ON GetStudentOrders TO student;
GRANT SELECT ON GetStudentResultsFromStudies TO student;
GRANT SELECT ON GetCourseModulesPassed TO student;
GRANT SELECT ON GetStudentResultsFromStudies TO student;
```

---

12. Rola: `guest`

Rola dla gości systemu.

Uprawnienia wykonawcze (EXECUTE):

- Dodawanie nowych użytkowników do systemu (`AddUser`).

```
CREATE ROLE guest;
GRANT EXECUTE ON AddUser TO guest;
```

---

13. Rola: `payment_system`

Rola techniczna dla systemu płatności. Szczegóły uprawnień mogą być dostosowane do potrzeb integracji z zewnętrznymi systemami finansowymi. Obecnie system płatności posiada możliwość modyfikacji zamówień w celu dodania statusu płatności po jej dokonaniu.

```
CREATE ROLE payment_system;
GRANT EXECUTE ON ModifyOrder TO payment_system;
```
