

PROJEKT - Podstawy Baz Danych

Nazwa bazy danych: u_sbarczyk

Autorzy:

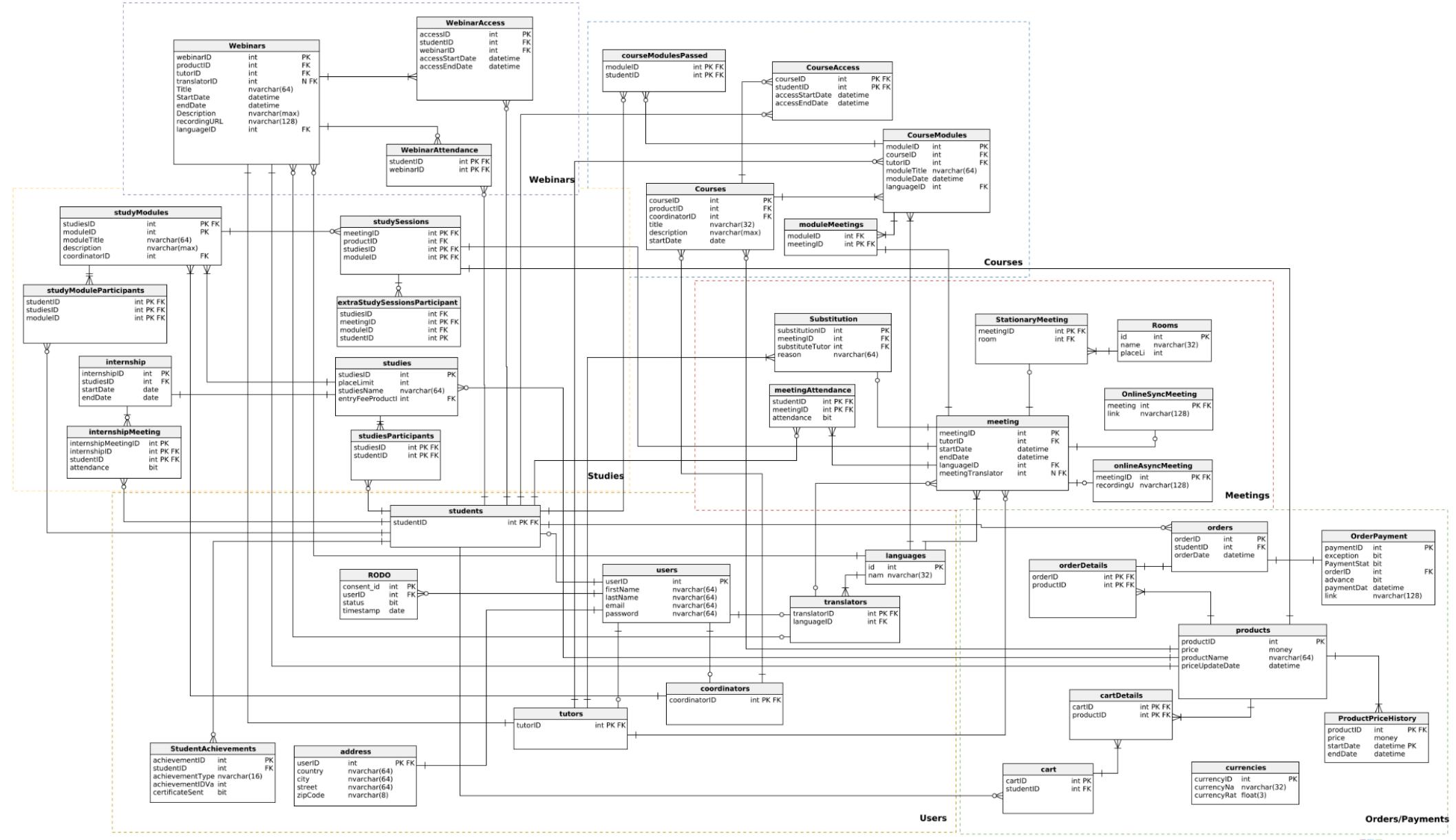
Szymon Barczyk - 33%: organizacja pracy, koncepcyjny projekt bazy danych wraz z jej schematem i warunkami integralności, widoki (implementacja), generowanie danych, funkcje (konsepcja), dokumentacja

Jan Dyląg - 33%: koncepcyjny projekt bazy danych wraz z jej schematem i warunkami integralności, procedury (implementacja), role, uprawnienia, indeksy, widoki (konsepcja), dokumentacja

Maciej Trznadel - 33%: koncepcyjny projekt bazy danych wraz z jej schematem i warunkami integralności, procedury (konsepcja), funkcje(implementacja), opisy tabel, trigerry, dokumentacja

Funkcje realizowane przez system zostały wykonane wspólnie.

SCHEMAT BAZY



OPISY TABEL

Sekcja Webinars

Webinars

Tabela przechowuje informacje o webinarach.

- webinarID - unikalne ID webinaru (PRIMARY KEY).
- productID - ID produktu (FOREIGN KEY)
- tutorID - ID prowadzącego webinar (FOREIGN KEY).
- translatorID - ID tłumacza webinaru (N, FOREIGN KEY).
- Title - tytuł webinaru.
- StartDate - data rozpoczęcia webinaru.
- Description - opis webinaru.
- recordingURL - link do nagrania webinaru.
- languageID - język webinaru (FOREIGN KEY).

```
CREATE TABLE Webinars (
    webinarID int NOT NULL IDENTITY,
    productID int NOT NULL,
    tutorID int NOT NULL,
    translatorID int NULL,
    Title nvarchar(64) NOT NULL,
    StartDate datetime NOT NULL,
    Description nvarchar(max) NOT NULL,
    recordingURL nvarchar(128) NOT NULL,
    languageID int NOT NULL,
    CONSTRAINT Webinars_pk PRIMARY KEY (webinarID)
);
```

```
ALTER TABLE Webinars ADD CONSTRAINT Webinars_languages
    FOREIGN KEY (languageID)
    REFERENCES languages (id);
```

```
ALTER TABLE Webinars ADD CONSTRAINT Webinars_translators
    FOREIGN KEY (translatorID)
    REFERENCES translators (translatorID);
```

```
ALTER TABLE Webinars ADD CONSTRAINT products_Webinars
    FOREIGN KEY (productID)
    REFERENCES products (productID);
```

```
ALTER TABLE Webinars ADD CONSTRAINT tutors_Webinars
    FOREIGN KEY (tutorID)
    REFERENCES tutors (tutorID);
```

WebinarAccess

Tabela przechowuje informacje o dostępie użytkowników do webinarów.

- accessID - unikalne ID dostępu (PRIMARY KEY).
- studentID - ID ucznia mającego dostęp do webinaru (FOREIGN KEY).
- webinarID - ID webinaru (FOREIGN KEY).
- accessStartDate - data rozpoczęcia dostępu.
- accessEndDate - data zakończenia dostępu.

Warunki integralności:

- accessEndDate musi być późniejsza niż accessStartDate, jeśli obie daty są obecne.

```
CREATE TABLE WebinarAccess (
    accessID int NOT NULL IDENTITY,
    studentID int NOT NULL,
    webinarID int NOT NULL,
    accessStartDate datetime NOT NULL,
    accessEndDate datetime NOT NULL,
    CONSTRAINT webinarAccessDateCheck CHECK (accessEndDate > accessStartDate),
    CONSTRAINT WebinarAccess_pk PRIMARY KEY (accessID)
);
```

```
ALTER TABLE WebinarAccess ADD CONSTRAINT WebinarAccess_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

```
ALTER TABLE WebinarAccess ADD CONSTRAINT WebinarDetails_Webinars
    FOREIGN KEY (webinarID)
    REFERENCES Webinars (webinarID);
```

WebinarAttendance

Tabela przechowuje informacje o uczestnikach, którzy byli obecni na Webinarze.

- studentID - unikalne ID studenta (PRIMARY KEY, FOREIGN KEY).
- webinarID - ID webinaru (PRIMARY KEY, FOREIGN KEY).

```
CREATE TABLE WebinarAttendance (
    studentID int NOT NULL,
    webinarID int NOT NULL,
    CONSTRAINT WebinarAttendance_pk PRIMARY KEY (studentID,webinarID)
);
```

```
ALTER TABLE WebinarAttendance ADD CONSTRAINT WebinarAttendance_Webinars
FOREIGN KEY (webinarID)
REFERENCES Webinars (webinarID);
```

```
ALTER TABLE WebinarAttendance ADD CONSTRAINT WebinarAttendance_students
FOREIGN KEY (studentID)
REFERENCES students (studentID);
```

Sekcja Courses

Courses

Tabela przechowuje informacje o kursach.

- courseID - unikalne ID kursu (PRIMARY KEY).
- productID - ID produktu - course (FOREIGN KEY)
- coordinatorID - ID koordynatora kursu (FOREIGN KEY).
- title - tytuł kursu.
- description - opis kursu.
- startDate - data rozpoczęcia kursu

```
CREATE TABLE Courses (
    courseID int NOT NULL IDENTITY,
    productID int NOT NULL,
    coordinatorID int NOT NULL,
    title nvarchar(32) NOT NULL,
    description nvarchar(max) NOT NULL,
    startDate date NOT NULL,
    CONSTRAINT Courses_pk PRIMARY KEY (courseID)
);
```

```
ALTER TABLE Courses ADD CONSTRAINT Courses_coordinators
    FOREIGN KEY (coordinatorID)
    REFERENCES coordinators (coordinatorID);
```

```
ALTER TABLE Courses ADD CONSTRAINT Courses_products
    FOREIGN KEY (productID)
    REFERENCES products (productID);
```

CourseAccess

Tabela przechowuje informacje o dostępie studentów do kursów.

- courseID - ID kursu (PRIMARY KEY, FOREIGN KEY).
- studentID - ID studenta (PRIMARY KEY, FOREIGN KEY).
- accessStartDate - data rozpoczęcia dostępu
- accessEndDate - data zakończenia dostępu .

Warunki integralności:

- accessEndDate musi być większa niż data rozpoczęcia dostępu accessStartDate

```
CREATE TABLE CourseAccess (
    courseID int NOT NULL,
    studentID int NOT NULL,
    accessStartDate datetime NOT NULL,
    accessEndDate datetime NOT NULL,
    CONSTRAINT accessDateCheck CHECK (accessEndDate > accessStartDate),
    CONSTRAINT CourseAccess_pk PRIMARY KEY (courseID,studentID)
);
```

```
ALTER TABLE CourseAccess ADD CONSTRAINT CourseAccess_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

```
ALTER TABLE CourseAccess ADD CONSTRAINT CourseDetails_Courses
    FOREIGN KEY (courseID)
    REFERENCES Courses (courseID);
```

CourseModules

Tabela przechowuje informacje o modułach wchodzących w skład kursów.

- moduleID - unikalne ID modułu (PRIMARY KEY).
- courseID - ID kursu, do którego należy moduł (FOREIGN KEY).
- tutorID - ID prowadzącego moduł (FOREIGN KEY).
- moduleTitle - tytuł modułu.
- moduleStartDate - data rozpoczęcia modułu.
- languageID - ID języka modułu (FOREIGN KEY).

```
CREATE TABLE CourseModules (
    moduleID int NOT NULL,
    courseID int NOT NULL,
    tutorID int NOT NULL,
    moduleTitle nvarchar(64) NOT NULL,
    moduleDate datetime NOT NULL,
    languageID int NOT NULL,
    CONSTRAINT CourseModules_pk PRIMARY KEY (moduleID)
);
```

```
ALTER TABLE CourseModules ADD CONSTRAINT CourseModules_Courses
    FOREIGN KEY (courseID)
    REFERENCES Courses (courseID);
```

```
ALTER TABLE CourseModules ADD CONSTRAINT CourseModules_languages
    FOREIGN KEY (languageID)
    REFERENCES languages (id);
```

```
ALTER TABLE CourseModules ADD CONSTRAINT CourseModules_tutors
    FOREIGN KEY (tutorID)
    REFERENCES tutors (tutorID);
```

courseModulesPassed

Tabela przechowuje informacje o studentach którzy zaliczyli moduł.

- moduleID - ID modułu (PRIMARY KEY, FOREIGN KEY).
- studentID - ID studenta, który zaliczył moduł (PRIMARY KEY, FOREIGN KEY).

```
CREATE TABLE courseModulesPassed (
    moduleID int NOT NULL,
    studentID int NOT NULL,
    CONSTRAINT courseModulesPassed_pk PRIMARY KEY (moduleID,studentID)
);
```

```
ALTER TABLE courseModulesPassed ADD CONSTRAINT courseModulesPassed_CourseModules
    FOREIGN KEY (moduleID)
    REFERENCES CourseModules (moduleID);
```

```
ALTER TABLE courseModulesPassed ADD CONSTRAINT courseModulesPassed_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

moduleMeetings

Tabela przechowuje informacje o spotkaniach związanych z modułami.

- moduleID - ID modułu, z którym jest powiązane spotkanie (FOREIGN KEY).
- meetingID - ID spotkania (PRIMARY KEY, FOREIGN KEY).
- Warunki integralności:
- Każdy moduleID musi istnieć w tabeli CourseModules (FOREIGN KEY).

```
CREATE TABLE moduleMeetings (
    moduleID int NOT NULL,
    meetingID int NOT NULL,
    CONSTRAINT moduleMeetings_pk PRIMARY KEY (meetingID)
);
```

```
ALTER TABLE moduleMeetings ADD CONSTRAINT meeting_moduleMeetings
    FOREIGN KEY (meetingID)
    REFERENCES meeting (meetingID);
```

```
ALTER TABLE moduleMeetings ADD CONSTRAINT moduleMeetings_CourseModules
    FOREIGN KEY (moduleID)
    REFERENCES CourseModules (moduleID);
```

Sekcja Studies

studies

Tabela przechowuje informacje o studiach.

- studiesID - unikalne ID studiów (PRIMARY KEY).
- placeLimit - maksymalna liczba miejsc dostępnych na dane studia.
- studiesName - nazwa studiów

Warunki integralności:

- placeLimit musi być większe lub równe 0.

```
CREATE TABLE studies (
    studiesID int NOT NULL IDENTITY,
    placeLimit int NOT NULL CHECK (placeLimit>0),
    CONSTRAINT studies_pk PRIMARY KEY (studiesID)
);
```

studyModules

Tabela przechowuje informacje o modułach przypisanych do studiów.

- studiesID - ID studiów, do których należy moduł (PRIMARY KEY, FOREIGN KEY).
- moduleID - unikalne ID modułu (PRIMARY KEY).
- moduleTitle - tytuł modułu
- description - opis modułu.
- coordinatorID - ID koordynatora modułu (FOREIGN KEY).

Warunki integralności:

- Każdy moduł musi być powiązany z istniejącymi studiami (FOREIGN KEY).

```
CREATE TABLE studyModules (
    studiesID int NOT NULL,
    moduleID int NOT NULL IDENTITY,
    moduleTitle nvarchar(64) NOT NULL,
    description nvarchar(max) NOT NULL,
    coordinatorID int NOT NULL,
    CONSTRAINT studyModules_pk PRIMARY KEY (studiesID, moduleID)
);
```

```
ALTER TABLE studyModules ADD CONSTRAINT studyModules_coordinators
    FOREIGN KEY (coordinatorID)
    REFERENCES coordinators (coordinatorID);
```

```
ALTER TABLE studyModules ADD CONSTRAINT study_modules_studies
    FOREIGN KEY (studiesID)
    REFERENCES studies (studiesID);
```

studyModuleParticipants

Tabela przechowuje informacje o uczestnikach modułów przypisanych do studiów.

- studentID - ID studenta (PRIMARY KEY, FOREIGN KEY).
- studiesID - ID studiów, do których przypisany jest uczestnik (PRIMARY KEY, FOREIGN KEY).
- moduleID - ID modułu studiów, w którym uczestniczy student (PRIMARY KEY, FOREIGN KEY).

Warunki integralności:

1. Kombinacja pól studentID, studiesID i moduleID musi być unikalna (klucz główny).
2. studentID, studiesID oraz moduleID muszą istnieć w odpowiadających im tabelach nadrzędnych jako klucze obce.

```
CREATE TABLE studyModuleParticipants (
    studentID int NOT NULL,
    studiesID int NOT NULL,
    moduleID int NOT NULL,
    CONSTRAINT studyModuleParticipants_pk PRIMARY KEY (studentID,studiesID,moduleID)
);
```

```
ALTER TABLE studyModuleParticipants ADD CONSTRAINT studyModuleParticipants_students
FOREIGN KEY (studentID)
REFERENCES students (studentID);
```

```
ALTER TABLE studyModuleParticipants ADD CONSTRAINT studyModuleParticipants_studyModules
FOREIGN KEY (studiesID,moduleID)
REFERENCES studyModules (studiesID,moduleID);
```

studySessions

Tabela przechowuje informacje o pojedynczych sesjach studiów.

- meetingID - unikalne ID spotkania (PRIMARY KEY, FOREIGN KEY).
- productID - ID produktu - studySessions (FOREIGN KEY)
- studiesID - ID studiów, do których należy sesja (PRIMARY KEY, FOREIGN KEY).
- moduleID - ID modułu, do którego należy sesja (PRIMARY KEY, FOREIGN KEY).

```
CREATE TABLE studySessions (
    meetingID int NOT NULL,
    productID int NOT NULL,
    studiesID int NOT NULL,
    moduleID int NOT NULL,
    CONSTRAINT studySessions_pk PRIMARY KEY (studiesID,meetingID,moduleID)
);
```

```
ALTER TABLE studySessions ADD CONSTRAINT products_studySessions
FOREIGN KEY (productID)
REFERENCES products (productID);
```

```
ALTER TABLE studySessions ADD CONSTRAINT studySessions_meeting
    FOREIGN KEY (meetingID)
    REFERENCES meeting (meetingID);
```

```
ALTER TABLE studySessions ADD CONSTRAINT studySessions_studyModules
    FOREIGN KEY (studiesID,moduleID)
    REFERENCES studyModules (studiesID,moduleID);
```

extraStudySessionsParticipants

Tabela przechowuje informacje o studentach, którzy zakupili dostęp do pojedynczego spotkania studyjnego.

- meetingID - unikalne ID spotkania (PRIMARY KEY, FOREIGN KEY).
- moduleID - ID modułu (PRIMARY KEY, FOREIGN KEY)
- studiesID - ID studiów, do których należy sesja (PRIMARY KEY, FOREIGN KEY).
- studentID - ID studenta, który zakupił dostęp do pojedynczej sesji studyjnej (PRIMARY KEY)

Warunki integralności:

- Student musi mieć opłacony dostęp do sesji studyjnej

```
CREATE TABLE extraStudySessionsParticipants (
    studiesID int NOT NULL,
    meetingID int NOT NULL,
    moduleID int NOT NULL,
    studentID int NOT NULL,
    CONSTRAINT extraStudySessionsParticipants_pk PRIMARY KEY (meetingID,studentID)
);
```

```
ALTER TABLE extraStudySessionsParticipants ADD CONSTRAINT extraStudySessionsParticipants_studySessions
    FOREIGN KEY (studiesID,meetingID,moduleID)
    REFERENCES studySessions (studiesID,meetingID,moduleID);
```

internship

Tabela przechowuje informacje o praktykach przypisanych do studiów.

- internshipID - unikalne ID praktyki (PRIMARY KEY).
- studiesID - ID studiów, do których należy praktyka (FOREIGN KEY).
- startDate - data rozpoczęcia praktyki.
- endDate - data zakończenia praktyk

Warunki integralności:

- startDate musi być wcześniejsza niż endDate

```
CREATE TABLE internship (
    internshipID int NOT NULL IDENTITY,
    studiesID int NOT NULL,
    startDate date NOT NULL,
    endDate date NOT NULL,
    CONSTRAINT internshipDateCheck CHECK (endDate > startDate),
    CONSTRAINT internship_pk PRIMARY KEY (internshipID)
);
```

```
ALTER TABLE internship ADD CONSTRAINT internship_studies
    FOREIGN KEY (studiesID)
    REFERENCES studies (studiesID);
```

internshipMeeting

Tabela przechowuje szczegółowe informacje o uczestnictwie studentów w praktykach.

- internshipMeetingID - ID spotkania w formie praktyk (PRIMARY KEY).
- internshipID - ID praktyki (PRIMARY KEY, FOREIGN KEY).
- studentID - ID studenta (PRIMARY KEY, FOREIGN KEY).
- attendance - informacja o obecności studenta na praktykach (wartość logiczna: 0 lub 1).

Warunki integralności:

- Każdy rekord musi odnosić się do istniejących praktyk i studentów (FOREIGN KEYS).

```
CREATE TABLE internshipMeeting (
    internshipMeetingID int NOT NULL IDENTITY,
    internshipID int NOT NULL,
    studentID int NOT NULL,
    attendance bit NOT NULL,
    CONSTRAINT internshipMeeting_pk PRIMARY KEY (internshipID,studentID,internshipMeetingID)
);
```

```
ALTER TABLE internshipMeeting ADD CONSTRAINT internshipDetails_internship
    FOREIGN KEY (internshipID)
    REFERENCES internship (internshipID);
```

```
ALTER TABLE internshipMeeting ADD CONSTRAINT internshipDetails_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

studiesParticipants

Tabela przechowuje informacje o studentach na danych studiach.

- studiesID - ID studiów, do których należy program (PRIMARY KEY, FOREIGN KEY).
- studentID - ID studenta (PRIMARY KEY, FOREIGN KEY)
- Warunki integralności:
- Każdy rekord musi być powiązany z istniejącymi studiów (FOREIGN KEY).

```
CREATE TABLE studiesParticipants (
    studiesID int NOT NULL,
    studentID int NOT NULL,
    CONSTRAINT studiesParticipants_pk PRIMARY KEY (studiesID,studentID)
);
```

```
ALTER TABLE studiesParticipants ADD CONSTRAINT studiesDetails_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

```
ALTER TABLE studiesParticipants ADD CONSTRAINT studiesDetails_studies
    FOREIGN KEY (studiesID)
    REFERENCES studies (studiesID);
```

Sekcja Meetings

meeting

Tabela przechowuje informacje o spotkaniach.

- meetingID - unikalne ID spotkania (PRIMARY KEY).
- tutorID - ID prowadzącego spotkanie (FOREIGN KEY).
- startDate - data i czas rozpoczęcia spotkania.
- endDate - data i czas zakończenia spotkania.
- languageID - ID języka spotkania (FOREIGN KEY).
- meetingTranslator - ID tłumacza przypisanego do spotkania (N, FOREIGN KEY)

Warunki integralności:

- endDate musi być późniejsza niż startDate.

```
CREATE TABLE meeting (
    meetingID int NOT NULL IDENTITY,
    tutorID int NOT NULL,
    startDate datetime NOT NULL,
    endDate datetime NOT NULL,
    languageID int NOT NULL,
    meetingTranslator int NULL,
    CONSTRAINT meetingDateCheck CHECK (endDate > startDate),
    CONSTRAINT meeting_pk PRIMARY KEY (meetingID)
);
```

```
ALTER TABLE meeting ADD CONSTRAINT meeting_languages
    FOREIGN KEY (languageID)
    REFERENCES languages (id);
```

```
ALTER TABLE meeting ADD CONSTRAINT meeting_translators
    FOREIGN KEY (meetingTranslator)
    REFERENCES translators (translatorID);
```

```
ALTER TABLE meeting ADD CONSTRAINT meeting_tutors
    FOREIGN KEY (tutorID)
    REFERENCES tutors (tutorID);
```

meetingAttendance

Tabela przechowuje informacje o obecności studentów na spotkaniach.

- studentID - ID studenta (PRIMARY KEY, FOREIGN KEY).
- meetingID - ID spotkania (PRIMARY KEY, FOREIGN KEY).
- attendance - status obecności studenta (wartość logiczna: 0 lub 1).

Warunki integralności:

- Kombinacja studentID i meetingID musi być unikalna (PRIMARY KEY).

```
CREATE TABLE meetingAttendance (
    studentID int NOT NULL,
    meetingID int NOT NULL,
    attendance bit NOT NULL DEFAULT 0,
    CONSTRAINT meetingAttendance_pk PRIMARY KEY (studentID,meetingID)
);
```

```
ALTER TABLE meetingAttendance ADD CONSTRAINT meetingPresence_meeting
    FOREIGN KEY (meetingID)
    REFERENCES meeting (meetingID);
```

```
ALTER TABLE meetingAttendance ADD CONSTRAINT pressence_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

Rooms

Tabela przechowuje informacje o salach, w których odbywają się spotkania stacjonarne.

- id - unikalne ID sali (PRIMARY KEY).
- name - nazwa sali.
- placeLimit - maksymalna liczba dostępnych w sali.

Warunki integralności:

- placeLimit musi być większe lub równe 0.

```
CREATE TABLE Rooms (
    id int NOT NULL IDENTITY,
    name nvarchar(32) NOT NULL,
    placeLimit int NOT NULL CHECK (placeLimit>0),
    CONSTRAINT Rooms_pk PRIMARY KEY (id)
);
```

StationaryMeeting

Tabela przechowuje informacje o spotkaniach stacjonarnych.

- meetingID - ID spotkania (PRIMARY KEY, FOREIGN KEY).
- room - ID sali, w której odbywa się spotkanie (FOREIGN KEY).

Warunki integralności:

- Każde spotkanie stacjonarne musi być powiązane z istniejącym rekordem w tabeli meeting.
- room musi odnosić się do istniejącej sali w tabeli Rooms.

```
CREATE TABLE StationaryMeeting (
    meetingID int NOT NULL,
    room int NOT NULL,
    CONSTRAINT StationaryMeeting_pk PRIMARY KEY (meetingID)
);
```

```
ALTER TABLE StationaryMeeting ADD CONSTRAINT StationaryMeeting_meeting
    FOREIGN KEY (meetingID)
    REFERENCES meeting (meetingID);
```

```
ALTER TABLE StationaryMeeting ADD CONSTRAINT StationaryModule_Rooms
    FOREIGN KEY (room)
    REFERENCES Rooms (id);
```

OnlineSyncMeeting

Tabela przechowuje informacje o synchronizowanych spotkaniach online.

- meetingID - ID spotkania (PRIMARY KEY, FOREIGN KEY).
- link - link do spotkania online.

Warunki integralności:

- Każde spotkanie online musi być powiązane z istniejącym rekordem w tabeli meeting.

```
CREATE TABLE OnlineSyncMeeting (
    meetingID int NOT NULL,
    link nvarchar(128) NOT NULL,
    CONSTRAINT OnlineSyncMeeting_pk PRIMARY KEY (meetingID)
);
```

```
ALTER TABLE OnlineSyncMeeting ADD CONSTRAINT OnlineSyncMeeting_meeting
    FOREIGN KEY (meetingID)
    REFERENCES meeting (meetingID);
```

onlineAsyncMeeting

Tabela przechowuje informacje o asynchronicznych spotkaniach online.

- meetingID - ID spotkania (PRIMARY KEY, FOREIGN KEY).
- recordingURL - link do nagrania spotkania.

Warunki integralności:

- Każde asynchroniczne spotkanie online musi być powiązane z istniejącym rekordem w tabeli meeting.

```
CREATE TABLE onlineAsyncMeeting (
    meetingID int NOT NULL,
    recordingURL nvarchar(128) NOT NULL,
    CONSTRAINT onlineAsyncMeeting_pk PRIMARY KEY (meetingID)
);
```

```
ALTER TABLE onlineAsyncMeeting ADD CONSTRAINT onlineAsyncMeeting_meeting
FOREIGN KEY (meetingID)
REFERENCES meeting (meetingID);
```

Substitution

Tabela przechowuje informacje o zastępstwach prowadzących spotkania.

- substitutionID - unikalne ID zastępstwa (PRIMARY KEY).
- meetingID - ID spotkania, którego dotyczy zastępstwo (FOREIGN KEY).
- substituteTutorID - ID zastępującego prowadzącego (FOREIGN KEY).
- reason - powód zastępstwa.

Warunki integralności:

- Każde zastępstwo musi być powiązane z istniejącym spotkaniem (FOREIGN KEY).

```
CREATE TABLE Substitution (
    substitutionID int NOT NULL IDENTITY,
    meetingID int NOT NULL,
    substituteTutorID int NOT NULL,
    reason nvarchar(64) NOT NULL,
    CONSTRAINT Substitution_pk PRIMARY KEY (substitutionID)
);
```

```
ALTER TABLE Substitution ADD CONSTRAINT Substitution_meeting
    FOREIGN KEY (meetingID)
    REFERENCES meeting (meetingID);
```

```
ALTER TABLE Substitution ADD CONSTRAINT Substitution_tutors
    FOREIGN KEY (substituteTutorID)
    REFERENCES tutors (tutorID);
```

Sekcja Users

users

Tabela przechowuje informacje o użytkownikach systemu.

- userID - unikalne ID użytkownika (PRIMARY KEY).
- firstName - imię użytkownika.
- lastName - nazwisko użytkownika.
- email - adres e-mail użytkownika.
- password - zaszyfrowane hasło użytkownika.

Warunki integralności:

- email musi być unikalny w ramach systemu.
- password powinno spełniać określone wymagania bezpieczeństwa (np. minimalna liczba znaków, złożoność).

```
CREATE TABLE users (
    userID int NOT NULL IDENTITY,
    firstName nvarchar(64) NOT NULL,
    lastName nvarchar(64) NOT NULL,
    email nvarchar(64) NOT NULL,
    password nvarchar(64) NOT NULL,
    CONSTRAINT email UNIQUE (email),
    CONSTRAINT users_pk PRIMARY KEY (userID)
);
```

students

Tabela przechowuje informacje o studentach, którzy są użytkownikami systemu.

- studentID - unikalne ID studenta (PRIMARY KEY, FOREIGN KEY powiązane z userID).

Warunki integralności:

- Każdy rekord w students musi odnosić się do istniejącego użytkownika w tabeli users (FOREIGN KEY).

```
CREATE TABLE students (
    studentID int NOT NULL,
    CONSTRAINT students_pk PRIMARY KEY (studentID)
);
```

```
ALTER TABLE students ADD CONSTRAINT students_users
    FOREIGN KEY (studentID)
    REFERENCES users (userID);
```

address

Tabela przechowuje adresy przypisane do użytkowników.

- userID - ID użytkownika(PRIMARY KEY, FOREIGN KEY).

- country - kraj zamieszkania.
- city - miasto zamieszkania.
- street - ulica zamieszkania.
- zipCode - kod pocztowy.
- Warunki integralności:
- userID musi odnosić się do istniejącego rekordu w tabeli user.
- zipCode powinien mieć odpowiedni format zależny od kraju.

```
CREATE TABLE address (
    userID int NOT NULL,
    country nvarchar(64) NOT NULL,
    city nvarchar(64) NOT NULL,
    street nvarchar(64) NOT NULL,
    zipCode nvarchar(8) NOT NULL CHECK (zipCode LIKE '[0-9][0-9]-[0-9][0-9][0-9]'),
    CONSTRAINT address_pk PRIMARY KEY (userID)
);
```

```
ALTER TABLE address ADD CONSTRAINT address_users
FOREIGN KEY (userID)
REFERENCES users (userID);
```

tutors

Tabela przechowuje informacje o prowadzących zajęcia.

- tutorID - unikalne ID prowadzącego (PRIMARY KEY, FOREIGN KEY powiązane z userID).

Warunki integralności:

- Każdy rekord w tutors musi odnosić się do istniejącego użytkownika w tabeli users (FOREIGN KEY).

```
CREATE TABLE tutors (
    tutorID int NOT NULL,
    CONSTRAINT tutors_pk PRIMARY KEY (tutorID)
);
```

```
ALTER TABLE tutors ADD CONSTRAINT tutors_users
FOREIGN KEY (tutorID)
REFERENCES users (userID);
```

coordinators

Tabela przechowuje informacje o koordynatorach.

- coordinatorID - unikalne ID koordynatora (PRIMARY KEY, FOREIGN KEY powiązane z userID).

Warunki integralności:

- Każdy rekord w coordinators musi odnosić się do istniejącego użytkownika w tabeli users (FOREIGN KEY).

```
CREATE TABLE coordinators (
    coordinatorID int NOT NULL,
    CONSTRAINT coordinators_pk PRIMARY KEY (coordinatorID)
);
```

```
ALTER TABLE coordinators ADD CONSTRAINT coordinators_users
    FOREIGN KEY (coordinatorID)
    REFERENCES users (userID);
```

translators

Tabela przechowuje informacje o tłumaczach.

- translatorID - unikalne ID tłumacza (PRIMARY KEY, FOREIGN KEY powiązane z userID).
- languagelID - ID języka, którym posługuje się tłumacz (FOREIGN KEY).

Warunki integralności:

- Każdy rekord w translators musi odnosić się do istniejącego użytkownika w tabeli users (FOREIGN KEY).
- languagelID musi odnosić się do istniejącego języka w tabeli languages.

```
CREATE TABLE translators (
    translatorID int NOT NULL,
    languageID int NOT NULL,
    CONSTRAINT translators_pk PRIMARY KEY (translatorID)
);
```

```
ALTER TABLE translators ADD CONSTRAINT translators_languages
    FOREIGN KEY (languageID)
    REFERENCES languages (id);
```

```
ALTER TABLE translators ADD CONSTRAINT translators_users
    FOREIGN KEY (translatorID)
    REFERENCES users (userID);
```

languages

Tabela przechowuje listę obsługiwanych języków.

- id - unikalne ID języka (PRIMARY KEY).
- name - nazwa języka.

Warunki integralności:

- name musi być unikalna w ramach tabeli.

```
CREATE TABLE Languages (
    id int NOT NULL,
    name nvarchar(32) NOT NULL,
    CONSTRAINT languages_pk PRIMARY KEY (id)
);
```

RODO

Tabela przechowuje informacje o zgodach RODO użytkowników.

- consent_id - unikalne ID zgody (PRIMARY KEY).
- userID - ID użytkownika, którego dotyczy zgoda (FOREIGN KEY).

- status - status zgody (wartość logiczna: 0 lub 1).
- timestamp - data i czas udzielenia lub cofnięcia zgody.

Warunki integralności:

- Każdy rekord musi odnosić się do istniejącego użytkownika w tabeli users (FOREIGN KEY).

```
CREATE TABLE RODO (
    consent_id int NOT NULL,
    userID int NOT NULL,
    status bit NOT NULL,
    timestamp date NOT NULL,
    CONSTRAINT RODO_pk PRIMARY KEY (consent_id)
);
```

```
ALTER TABLE RODO ADD CONSTRAINT Table_73_users
FOREIGN KEY (userID)
REFERENCES users (userID);
```

StudentAchievements

Tabela przechowuje informacje o kursach/studiach, które ukończyli studenci, przechowuje również informacje czy dyplom ukończenia został wysłany.

- achievementID - unikalne ID osiągnięcia
- studentID - ID studenta
- achievementType - nazwa osiągnięcia Course lub Studies
- achievementIDValue - id kursu lub studiów
- certificateSent - czy certyfikat został wysłany (wartość logiczna 0 lub 1- jeśli został wysłany)

```
CREATE TABLE StudentAchievements (
    achievementID int NOT NULL IDENTITY,
    studentID int NOT NULL,
    achievementType nvarchar(16) NOT NULL,
    achievementIDValue int NOT NULL,
    certificateSent bit NOT NULL,
    CONSTRAINT StudentAchievements_pk PRIMARY KEY (achievementID)
);
```

```
ALTER TABLE StudentAchievements ADD CONSTRAINT StudentAchievements_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

Sekcja Orders/Payments

orders

Tabela przechowuje informacje o zamówieniach.

- orderID - unikalne ID zamówienia (PRIMARY KEY).
- studentID - ID studenta, który złożył zamówienie (FOREIGN KEY).
- orderDate - data złożenia zamówienia.

Warunki integralności:

- Każde zamówienie musi być przypisane do istniejącego studenta (FOREIGN KEY).

```
CREATE TABLE orders (
    orderID int NOT NULL IDENTITY,
    studentID int NOT NULL,
    orderDate datetime NOT NULL,
    CONSTRAINT orders_pk PRIMARY KEY (orderID)
);
```

```
ALTER TABLE orders ADD CONSTRAINT orders_students
    FOREIGN KEY (studentID)
    REFERENCES students (studentID);
```

orderDetails

Tabela przechowuje szczegóły zamówienia.

- orderID - ID zamówienia (PRIMARY KEY, FOREIGN KEY).
- productID - ID produktu, który jest częścią zamówienia (PRIMARY KEY, FOREIGN KEY).

Warunki integralności:

- Każdy rekord musi być powiązany z istniejącym zamówieniem (orderID) i produktem (productID).

```
CREATE TABLE orderDetails (
    orderID int NOT NULL,
    productID int NOT NULL,
    CONSTRAINT orderDetails_pk PRIMARY KEY (orderID,productID)
);
```

```
ALTER TABLE orderDetails ADD CONSTRAINT orderDetails_orders
    FOREIGN KEY (orderID)
    REFERENCES orders (orderID);
```

```
ALTER TABLE orderDetails ADD CONSTRAINT orderDetails_products
    FOREIGN KEY (productID)
    REFERENCES products (productID);
```

OrderPayment

Tabela przechowuje informacje o płatnościach za zamówienia.

- paymentID - unikalne ID płatności (PRIMARY KEY).
- exception - flaga oznaczająca wyjątek w płatności (wartość logiczna: 0 lub 1, gdy zastosowano wyjątek).
- PaymentStatus - status płatności (wartość logiczna: 0 = nieopłacone, 1 = opłacone).
- orderID - unikalne ID zamówienia (FOREIGN KEY)
- advance - zaliczka (wartość logiczna: 0 lub 1, gdy opłacona)

- paymentDate - data opłacenia zamówienia
- link - link do zewnętrznego serwisu płatności

Warunki integralności:

- Każda płatność musi być przypisana do istniejącego studenta i kursu (FOREIGN KEYS).

```
CREATE TABLE OrderPayment (
    paymentID int NOT NULL IDENTITY,
    exception bit NOT NULL,
    PaymentStatus bit NOT NULL,
    orderID int NOT NULL,
    advance bit NOT NULL DEFAULT 0,
    paymentDate datetime NOT NULL,
    CONSTRAINT OrderPayment_pk PRIMARY KEY (paymentID)
);
```

```
ALTER TABLE OrderPayment ADD CONSTRAINT coursePayment_orders
    FOREIGN KEY (orderID)
    REFERENCES orders (orderID);
```

ProductPriceHistory

Tabela przechowująca historie cen produktów

- productID - unikalne ID produktu (PRIMARY KEY, FOREIGN KEY).
- price - dawna cena produktu.
- startDate - data kiedy zaczęła obowiązywać cena
- endDate - data kiedy przestała obowiązywać cena

Warunki integralności:

Początkowa data aktualizacji ceny produktu nie może być większa od końcowej

```
CREATE TABLE ProductPriceHistory (
    productID int NOT NULL,
    price money NOT NULL CHECK (price>=0),
    startDate datetime NOT NULL,
    endDate datetime NOT NULL,
    CONSTRAINT priceHistoryDateCheck CHECK (endDate > startDate),
    CONSTRAINT ProductPriceHistory_pk PRIMARY KEY (productID)
);
```

```
ALTER TABLE ProductPriceHistory ADD CONSTRAINT ProductPriceHistory_products
    FOREIGN KEY (productID)
    REFERENCES products (productID);
```

products

Tabela przechowuje informacje o produktach.

- productID - unikalne ID produktu (PRIMARY KEY).
- price - cena produktu.
- productName - nazwa produktu.
- priceUpdateDate - data aktualizacji ceny

Warunki integralności:

- price musi być nieujemna (≥ 0).

```
CREATE TABLE products (
    productID int NOT NULL IDENTITY,
    price money NOT NULL CHECK (price>=0),
    productName nvarchar(64) NOT NULL,
    priceUpdateDate datetime NOT NULL,
    CONSTRAINT products_pk PRIMARY KEY (productID)
);
```

currencies

Tabela przechowuje informacje o kursach walutowych

- currencyID - unikalne ID kursu walutowego (PRIMARY KEY).
- currencyName - nazwa kursu walutowego.
- currencyRate - kurs walutowy.

```
CREATE TABLE currencies (
    currencyID int NOT NULL IDENTITY,
    currencyName nvarchar(32) NOT NULL,
    currencyRate float(3) NOT NULL,
    CONSTRAINT currencies_pk PRIMARY KEY (currencyID)
);
```

cart

Tabela przechowuje informacje o koszykach użytkowników.

- cartID - unikalne ID koszyka (PRIMARY KEY).
- studentID - ID studenta, który posiada koszyk (FOREIGN KEY).

Warunki integralności:

- Każdy koszyk musi być przypisany do istniejącego studenta (FOREIGN KEY).

```
CREATE TABLE cart (
    cartID int NOT NULL IDENTITY,
    studentID int NOT NULL,
    CONSTRAINT cart_pk PRIMARY KEY (cartID)
);
```

```
ALTER TABLE cart ADD CONSTRAINT cart_students
FOREIGN KEY (studentID)
REFERENCES students (studentID);
```

cartDetails

Tabela przechowuje szczegóły dotyczące produktów w koszyku.

- cartID - ID koszyka (PRIMARY KEY, FOREIGN KEY).
- productID - ID produktu w koszyku (PRIMARY KEY, FOREIGN KEY).

Warunki integralności:

- Każdy rekord musi być powiązany z istniejącym koszykiem i produktem.

```
CREATE TABLE cartDetails (
    cartID int NOT NULL,
    productID int NOT NULL,
    CONSTRAINT cartDetails_pk PRIMARY KEY (cartID,productID)
);
```

```
ALTER TABLE cartDetails ADD CONSTRAINT cartDetails_cart
    FOREIGN KEY (cartID)
    REFERENCES cart (cartID);
```

```
ALTER TABLE cartDetails ADD CONSTRAINT cartDetails_products
    FOREIGN KEY (productID)
    REFERENCES products (productID);
```

FUNKCJE REALIZOWANE PRZEZ SYSTEM

Sekcja: Users

(Zarządzanie użytkownikami i kontami)

1. Rejestracja nowego konta

- **Dostęp:** każdy potencjalny użytkownik (poza systemem) może się zarejestrować i zostać user + ewentualnie inną rolą.
- **Opis:** tworzy wpis w tabeli users, wypełnia dane adresowe, akceptuje RODO.

2. Logowanie do systemu

- **Dostęp:** wszyscy zalogowani (rola user, student, tutor, coordinator, translator).
- **Opis:** weryfikacja poświadczeń (e-mail, hasło) i przypisanie ról.

3. Aktualizacja danych konta (np. adresu)

- **Dostęp:**

- student: może edytować **swoje** dane (np. procedura studentModifyAddress).
- tutor, translator: najczęściej brak potrzeby edycji danych osobowych (lub ograniczony dostęp).
- coordinator: może modyfikować dane użytkowników w razie potrzeby organizacyjnej.

- **Opis:** zapisuje zmiany w tabeli users / address.

4. Dodawanie nowych użytkowników / usuwanie / zmiana ról

- **Dostęp:** coordinator (przyjęte jako osoba o uprawnieniach administracyjnych)
- **Opis:** koordynator może tworzyć konta np. innych tutorów / tłumaczy, usuwać zbędne konta, nadawać/updatować role.

5. Przegląd listy użytkowników z podziałem na role

- **Dostęp:** coordinator (pełny wgląd w bazę), tutor (może widzieć np. listy studentów).
- **Opis:** generowanie widoku / raportu z tabel users i ról (students, tutors, translators, coordinators).

Sekcja: Meetings

(Zarządzanie spotkaniami)

1. **Tworzenie nowego spotkania** (stacjonarnego, online lub asynchronicznego)
 - **Dostęp:** coordinator (jako osoba administrująca harmonogram).
 - **Opis:** wypełnia tabelę meeting, dodaje ewentualny wpis do StationaryMeeting / OnlineSyncMeeting / onlineAsyncMeeting.
2. **Edytowanie szczegółów spotkania** (zmiana daty, miejsca, języka, tłumacza)
 - **Dostęp:** coordinator i (częściowo) tutor.
 - **Opis:** tutor może wnioskować o zmianę terminu, ale ostatecznie coordinator zatwierdza. W tabelach meeting, StationaryMeeting, itp.
3. **Przypisywanie sal i platform**
 - **Dostęp:** coordinator.
 - **Opis:** zarządza Rooms (sale) i linkami do spotkań online.
4. **Podgląd harmonogramu spotkań**
 - **Dostęp:**

- student: widzi spotkania, na które jest zapisany (poprzez tabele meetingAttendance, studySessions, WebinarAttendance itp.).
 - tutor: widzi wszystkie spotkania, które prowadzi.
 - coordinator: pełen wgląd.
5. **Zarządzanie listą obecności** (np. dla spotkań stacjonarnych i synchronicznych online)
 - **Dostęp:** tutor (poprzez tabele meetingAttendance).
 - **Opis:** tutor może dodać / zaktualizować wpisy obecności w meetingAttendance.
 6. **Powiadamianie uczestników o zmianach w harmonogramie**
 - **Dostęp:** coordinator.
 - **Opis:** wysyła komunikaty do studentów (może być przez system zewnętrzny).
-

Sekcja: Orders / Payments

(Zarządzanie zamówieniami i płatnościami)

1. **Dodawanie produktów do koszyka**
 - **Dostęp:** student (podobnie user w roli klienta).
 - **Opis:** wypełnia tabele cart i cartDetails.
2. **Podgląd i edycja koszyka**
 - **Dostęp:** student (tylko swoich koszyków).
 - **Opis:** usuwanie / dodawanie pozycji w cartDetails.
3. **Finalizacja zamówienia i link do płatności**
 - **Dostęp:** student.
 - **Opis:** tworzy orders, orderDetails, generuje rekord w OrderPayment. Płatności obsługuje system zewnętrzny.
4. **Przegląd historii zamówień i statusów płatności**
 - **Dostęp:**
 - student: widzi swoje zamówienia.
 - coordinator: ma wgląd we wszystkie (np. do raportów).
 - tutor, translator: zwykle brak potrzeby dostępu.
5. **Generowanie raportów finansowych** (webinary, kursy, studia)
 - **Dostęp:** coordinator.
 - **Opis:** łączy dane z orders, orderDetails, OrderPayment, generuje widoki.
6. **Identyfikowanie dłużników** (nieopłacone zamówienia)

- **Dostęp:** coordinator.
 - **Opis:** sprawdza statusy w OrderPayment (np. brak płatności).
-

Sekcja: Studies

(Zarządzanie studiami)

1. **Tworzenie nowego programu studiów (syllabus)**
 - **Dostęp:** coordinator.
 - **Opis:** wypełnia tabelę studies (i produkt Wpisowe : ...), ewentualnie studyModules.
 2. **Edytowanie harmonogramu spotkań w ramach studiów**
 - **Dostęp:** coordinator, ewentualnie tutor w ograniczonym zakresie.
 - **Opis:** zmiany w studySessions (przypisanie meetingID, moduleID).
 3. **Zarządzanie listą studentów (przypisanych do studiów)**
 - **Dostęp:** coordinator.
 - **Opis:** tabele studiesParticipants, studyModuleParticipants, sprawdzanie limitów miejsc.
 4. **Frekwencja i obecności na studiach**
 - **Dostęp:**
 - tutor: wstawia obecność w meetingAttendance.
 - student: przegląda swój status (funkcja/raport).
 - **Opis:** generowane raporty z listą obecności.
 5. **Praktyki studenckie**
 - **Dostęp:** tutor i coordinator (ustalanie, potwierdzanie praktyk).
 - **Opis:** tabele internship i internshipMeeting (100% frekwencji wymagane).
 6. **Zapisy na pojedyncze spotkania w ramach studiów**
 - **Dostęp:** student.
 - **Opis:** tabele extraStudySessionsParticipants.
 7. **Generowanie dyplomów po ukończeniu studiów**
 - **Dostęp:** coordinator (wysyła do studentów).
 - **Opis:** tabele StudentAchievements.
-

Sekcja: Webinars

(Zarządzanie webinarami)

1. **Tworzenie nowego webinaru**
 - **Dostęp:** coordinator.
 - **Opis:** generuje Webinars (pozwala przypisać tutorID, ewentualnie translatorID).
2. **Zarządzanie szczegółami webinaru** (tematyka, data, link, nagranie)
 - **Dostęp:** coordinator.
 - **Opis:** wypełnia / zmienia w Webinars (np. link do recordingURL).
3. **Podgląd listy zapisanych uczestników**
 - **Dostęp:**
 - student: widzi **tylko** swoje webinary.
 - tutor: może mieć wgląd w wszystkich uczestników prowadzonych webinarów.
 - coordinator: pełny wgląd.
4. **Zarządzanie dostępem do webinaru**
 - **Dostęp:** coordinator.
 - **Opis:** po opłaceniu do WebinarAccess.
5. **Generowanie raportów finansowych** dla webinarów
 - **Dostęp:** coordinator.
6. **Usuwanie webinarów**
 - **Dostęp:** coordinator.
7. **Podgląd nagrani webinarów**
 - **Dostęp:** student (jeśli ma WebinarAccess), tutor, translator – w zakresie przypisanych webinarów.

Sekcja: Courses

(Zarządzanie kursami)

1. **Tworzenie nowego kursu**
 - **Dostęp:** coordinator.
 - **Opis:** tworzy wpis w Courses, przypisuje productID, koordynatora, opis.
2. **Zarządzanie modułami kursu** (stacjonarne, online, hybrydowe)
 - **Dostęp:** tutor (dla swojego kursu), coordinator.
 - **Opis:** tabele CourseModules, ewentualnie powiązane spotkania w moduleMeetings.
3. **Podgląd listy zapisanych uczestników**
 - **Dostęp:**

- student: widzi **swoje** kursy (przez CourseAccess, courseModulesPassed).
 - tutor: wgląd w uczestników prowadzonych kursów.
 - coordinator: pełen wgląd.
4. **Zarządzanie limitami miejsc** (dla kursów hybrydowych/stacjonarnych)
 - **Dostęp:** coordinator.
 5. **Podgląd statusu zaliczeń modułów**
 - **Dostęp:**
 - student: widzi **swoje** zaliczenia w courseModulesPassed.
 - tutor: pełny wgląd (dla prowadzonych kursów).
 6. **Zarządzanie dyplomami ukończenia kursu**
 - **Dostęp:** coordinator (wysyła do uczestników po spełnieniu warunków).
 7. **Powiadamianie uczestników o zmianach w harmonogramie**
 - **Dostęp:** coordinator.

FUNKCJE

Funkcje wspierające obsługę webinarów

1. userHasAccessToWebinar

Opis:

Sprawdzi, czy dany student (@studentID) ma aktualny dostęp do webinaru (@webinarID) na podstawie tabel WebinarAccess oraz Webinars, korzystając pomocnie z tabeli Products.

Najpierw w Webinars sprawdzimy, czy webinar jest płatny czy darmowy. Za pomocą productID sprawdzamy cenę tego webinaru w tabeli products (price = 0).

Jeśli darmowy (price = 0), wystarczy posiadanie konta (jeśli studentID jest w students) i wówczas dostęp jest natychmiastowy.

Jeśli płatny (price > 0), w WebinarAccess sprawdzimy, czy istnieje rekord z danym studentID i webinarID, a także czy AccessEndDate >= @currentDate.

Wejście:

- @studentID (int)
- @webinarID (int)
- @currentDate (date)

Wyjście:

- bit (1 – ma dostęp, 0 – brak dostępu)

```
CREATE FUNCTION userHasAccessToWebinar (
    @studentID INT,
    @webinarID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @isPaid BIT;
    DECLARE @hasAccess BIT;
    DECLARE @currentDate DATETIME = GETDATE();

    SELECT @isPaid = CASE
        WHEN p.price > 0 THEN 1
        ELSE 0
    END
    FROM Webinars w
    JOIN products p 1..n<->1: ON w.productID = p.productID
    WHERE w.webinarID = @webinarID;

    IF @isPaid = 0
    BEGIN
        IF EXISTS (
            SELECT 1
            FROM students
            WHERE studentID = @studentID
        )
        BEGIN
            SET @hasAccess = 1;
        END
        ELSE
        BEGIN
            SET @hasAccess = 0;
        END
    END
    ELSE

```

```

BEGIN
    IF EXISTS (
        SELECT 1
        FROM WebinarAccess wa
        WHERE wa.studentID = @studentID
            AND wa.webinarID = @webinarID
            AND wa.AccessEndDate >= @currentDate
    )
    BEGIN
        SET @hasAccess = 1;
    END
    ELSE
    BEGIN
        SET @hasAccess = 0;
    END
END

RETURN @hasAccess;
END
go

```

2. remainingWebinarAccessDays

Opis:

Oblicza liczbę pozostałych dni dostępu do opłaconego webinaru.

Z tabeli WebinarAccess pobiera AccessEndDate dla danego studentID i webinarID. Oblicza różnicę dni między @currentDate a AccessEndDate. Jeśli wynik < 0, zwraca 0, jeśli ≥0, zwraca liczbę dni.

Wejście:

- @studentID (int)

- @webinarID (int)
- @currentDate (date)

Wyjście:

- int (liczba dni – jeśli brak dostępu lub dostęp wygasł, może zwrócić 0)

```

CREATE FUNCTION remainingWebinarAccessDays (
    @studentID INT,
    @webinarID INT
)
RETURNS INT
AS
BEGIN
    DECLARE @remainingDays INT = 0;
    DECLARE @currentDate DATETIME = GETDATE();

    IF dbo.UserHasAccessToWebinar( @studentID @studentID, @webinarID @webinarID) = 0
    BEGIN
        RETURN @remainingDays;
    END

    SELECT @remainingDays = DATEDIFF(DAY, @currentDate, wa.AccessEndDate)
    FROM WebinarAccess wa
    WHERE wa.studentID = @studentID
        AND wa.webinarID = @webinarID;

    IF @remainingDays < 0
    BEGIN
        SET @remainingDays = 0;
    END

    RETURN @remainingDays;
END
go

grant execute on dbo.remainingWebinarAccessDays to Participant
go

```

3. webinarIsPaid

Opis:

Sprawdza, czy dany webinar (@webinarID) jest płatny.

W tabeli Webinars przez productID dla danego webinaru łączy tabelę z products i sprawdza cenę produktu. Jeśli price > 0 => 1, w przeciwnym razie 0.

Wejście:

- @webinarID (int)

Wyjście:

- bit (1 – płatny, 0 – darmowy)

```
CREATE FUNCTION webinarIsPaid (
    @webinarID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @isPaid BIT;

    SELECT @isPaid = CASE
        WHEN p.price > 0 THEN 1
        ELSE 0
    END
    FROM Webinars w
    JOIN products p 1..n<->1: ON w.productID = p.productID
    WHERE w.webinarID = @webinarID;

    RETURN @isPaid;
END
go
```

Funkcje wspierające obsługę kursów

4. userCourseAttendancePercentage

Opis:

Oblicza procent zaliczenia kursu przez studenta (@studentID) na podstawie liczby zaliczonych modułów.

Najpierw liczy całkowitą liczbę modułów w CourseModules przypisanych do courseID, a następnie liczbę zaliczonych modułów z tabeli courseModulesPassed. Na tej podstawie oblicza (zaliczone/ogółem) * 100.

Wejście:

- @courseID (int)
- @studentID (int)

Wyjście:

- decimal(5,2) – procent zaliczenia

```
CREATE FUNCTION userCourseAttendancePercentage (
    @courseID INT,
    @studentID INT
)
RETURNS DECIMAL(5, 2)
AS
BEGIN
    DECLARE @totalModules INT;
    DECLARE @passedModules INT;
    DECLARE @attendancePercentage DECIMAL(5, 2);

    SELECT @totalModules = COUNT(*)
    FROM CourseModules
    WHERE courseID = @courseID;

    SELECT @passedModules = COUNT(*)
    FROM CourseModules cm
    JOIN courseModulesPassed cmp 1<->1..n: ON cm.moduleID = cmp.moduleID
    WHERE cm.courseID = @courseID
        AND cmp.studentID = @studentID;

    IF @totalModules > 0
    BEGIN
        SET @attendancePercentage = CAST(@passedModules AS DECIMAL(5, 2)) / @totalModules * 100;
    END
    ELSE
    BEGIN
        SET @attendancePercentage = 0;
    END

    RETURN @attendancePercentage;
END
```

5. coursesPassedByUser

Opis:

Sprawdza, czy użytkownik zaliczył kurs, przyjmując kryterium minimum 80% zaliczonych modułów.

Korzysta z logiki z funkcji userCourseAttendancePercentage. Jeśli wynik ≥ 80 , zwraca 1, inaczej 0.

Wejście:

- @courseID (int)
- @studentID (int)

Wyjście:

- bit (1 – zaliczone, 0 – niezaliczone)

```
CREATE FUNCTION courseIsPassedByUser (
    @courseID INT,
    @studentID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @attendancePercentage DECIMAL(5, 2);
    DECLARE @isPassed BIT;

    SELECT @attendancePercentage = dbo.UserCourseAttendancePercentage( @courseID @courseID, @studentID @studentID);

    IF @attendancePercentage >= 80
    BEGIN
        SET @isPassed = 1;
    END
    ELSE
    BEGIN
        SET @isPassed = 0;
    END

    RETURN @isPassed;
END
```

6. CourseFeeFullyPaid

Opis:

Sprawdza, czy użytkownik opłacił kurs w całości, biorąc pod uwagę zasady (zaliczka przy zapisie, reszta kwoty 3 dni przed startem kursu).

Funkcja w orders, orderDetails oraz products sprawdza płatności przypisane do danego studentID i produktu związanego z danego courseId. Weryfikuje, czy zaliczka i kurs zostały opłacone we właściwym terminie, porównując daty płatności z datą rozpoczęcia kursu (Courses.startDate).

Wejście:

- @courseID (int)
- @studentID (int)

Wyjście:

- bit

```

CREATE FUNCTION CourseFeeFullyPaid (
    @courseID INT,
    @studentID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @isFullyPaid BIT = 0;
    DECLARE @startDate DATE;

    SELECT @startDate = startDate
    FROM Courses
    WHERE courseID = @courseID;

    IF EXISTS (
        SELECT 1
        FROM Orders o
        JOIN OrderPayment op 1<->1..n: ON o.orderID = op.orderID
        JOIN OrderDetails od 1<->1..n: ON o.orderID = od.orderID
        JOIN Products p 1..n<->1: ON od.productID = p.productID
        WHERE o.studentID = @studentID
            AND p.productID = @courseID
            AND op.advance = 1
            AND op.PaymentStatus = 1
            AND op.paymentDate <= DATEADD(DAY, -3, @startDate)
    )
    BEGIN
        SET @isFullyPaid = 1;
    END

    RETURN @isFullyPaid;
END

```

Funkcje wspierające obsługę studiów

7. studyAttendancePercentage

Opis:

Oblicza procent obecności studenta na studiach (@studiesID), uwzględniając liczbę wymaganych zajęć i liczbę zajęć, na których był obecny.

Na podstawie tabel studyModuleParticipants, studyModules, studySessions, meetings i meetingAttendance oblicza stosunek liczby obecności do liczby wszystkich wymaganych spotkań, a następnie wynik procentowy.

Wejście:

- @studiesID (int)
- @studentID (int)

Wyjście:

- decimal(5,2)

```

CREATE FUNCTION studyAttendancePercentage (
    @studiesID INT,
    @studentID INT
)
RETURNS DECIMAL(5, 2)
AS
BEGIN
    DECLARE @totalMeetings INT;
    DECLARE @attendedMeetings INT;
    DECLARE @attendancePercentage DECIMAL(5, 2) = 0;

    SELECT @totalMeetings = COUNT(DISTINCT m.meetingID)
    FROM studyModuleParticipants smp
    JOIN studyModules sm 1..n<->1: ON smp.studiesID = sm.studiesID AND smp.moduleID = sm.moduleID
    JOIN studySessions ss 1<->1..n: ON sm.studiesID = ss.studiesID AND sm.moduleID = ss.moduleID
    JOIN meeting m 1..n<->1: ON ss.meetingID = m.meetingID
    WHERE smp.studiesID = @studiesID AND smp.studentID = @studentID;

    SELECT @attendedMeetings = COUNT(DISTINCT ma.meetingID)
    FROM studyModuleParticipants smp
    JOIN studyModules sm 1..n<->1: ON smp.studiesID = sm.studiesID AND smp.moduleID = sm.moduleID
    JOIN studySessions ss 1<->1..n: ON sm.studiesID = ss.studiesID AND sm.moduleID = ss.moduleID
    JOIN meeting m 1..n<->1: ON ss.meetingID = m.meetingID
    JOIN meetingAttendance ma 1<->1..n: ON m.meetingID = ma.meetingID
    WHERE smp.studiesID = @studiesID
        AND smp.studentID = @studentID
        AND ma.studentID = @studentID
        AND ma.attendance = 1;

```

```

IF @totalMeetings > 0
BEGIN
    SET @attendancePercentage = CAST(@attendedMeetings AS DECIMAL(5, 2)) / @totalMeetings * 100;
END

RETURN @attendancePercentage;
END
go

grant execute on dbo.studyAttendancePercentage to Lecturer
go

```

8.studentHasCompletedInternship

Opis:

Sprawdza, czy student zaliczył praktyki (100% frekwencja przez 14 dni).

Funkcja w tabelach internship i internshipMeeting sprawdza, czy studentID uczestniczył w praktykach i ile ma wpisów attendance = 1. Jeśli ma co najmniej 14 wpisów, zwraca 1, w przeciwnym razie 0.

Wejście:

- @studiesID (int)
- @studentID (int)
- @internshipID (int)

Wyjście:

- bit

```
CREATE FUNCTION studentHasCompletedInternship (
    @studiesID INT,
    @studentID INT,
    @internshipID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @attendanceCount INT;
    DECLARE @hasCompleted BIT = 0;

    SELECT @attendanceCount = COUNT(*)
    FROM internshipMeeting im
    JOIN internship i 1..n<->1: ON im.internshipID = i.internshipID
    WHERE i.studiesID = @studiesID
        AND im.internshipID = @internshipID
        AND im.studentID = @studentID
        AND im.attendance = 1;

    IF @attendanceCount >= 14
    BEGIN
        SET @hasCompleted = 1;
    END

    RETURN @hasCompleted;
END
go
```

9. studySessionsFeePaid

Opis:

Sprawdza, czy student uiścił opłate za sesje studyjną.

Funkcja sprawdza w orders i orderDetails, payments, czy studentID opłacił produkt spotkanie powiązane z danymi studiami.

Wejście:

- @studiesID (int)
- @meetingID (int)
- @moduleID(int)
- @studyID(int)

Wyjście:

- bit

```

CREATE FUNCTION studySessionsFeePaid (
    @studiesID INT,
    @meetingID INT,
    @moduleId INT,
    @studentID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @isPaid BIT = 0;

    IF EXISTS (
        SELECT 1
        FROM studySessions ss
        JOIN products p 1..n<->1: ON ss.productID = p.productID
        JOIN orderDetails od 1<->1..n: ON p.productID = od.productID
        JOIN orders o 1..n<->1: ON od.orderID = o.orderID
        JOIN OrderPayment op 1<->1..n: ON o.orderID = op.orderID
        WHERE ss.meetingID = @meetingID
            AND ss.studiesID = @studiesID
            AND ss.moduleID = @moduleId
            AND o.studentID = @studentID
            AND op.PaymentStatus = 1
    )
    BEGIN
        SET @isPaid = 1;
    END

    RETURN @isPaid;
END
go

```

10. studySessionFeePaidOnTime

Opis:

Sprawdza, czy opłata za dany zjazd studiów (meetingID) została uregulowana 3 dni przed jego rozpoczęciem.

Funkcja w orders i orderDetails i payments sprawdza, czy student opłacił produkt powiązany z

meetingID lub studiesID i czy płatność została dokonana minimum 3 dni przed meeting.startDate.

Wejście:

- @meetingID (int)
- @studentID (int)
- @moduleID(int)
- @studyID(int)
-

Wyjście:

- bit

```

CREATE FUNCTION studySessionFeePaidOnTime (
    @meetingID INT,
    @studentID INT,
    @moduleID INT,
    @studyID INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @isPaidOnTime BIT = 0;

    -- Sprawdź, czy płatność została dokonana na czas
    IF EXISTS (
        SELECT 1
        FROM studySessions ss
        JOIN products p 1..n<->1: ON ss.productID = p.productID
        JOIN orderDetails od 1<->1..n: ON p.productID = od.productID
        JOIN orders o 1..n<->1: ON od.orderID = o.orderID
        JOIN OrderPayment op 1<->1..n: ON o.orderID = op.orderID
        JOIN meeting m 1..n<->1: ON ss.meetingID = m.meetingID
        WHERE ss.meetingID = @meetingID
            AND ss.studiesID = @studyID
            AND ss.moduleID = @moduleID
            AND o.studentID = @studentID
            AND op.PaymentStatus = 1
            AND op.paymentDate <= DATEADD(DAY, -3, m.startDate)
    )
    BEGIN
        SET @isPaidOnTime = 1;
    END

    RETURN @isPaidOnTime;
END
go

```

Funkcje wspierające limity miejsc

11. availableSeatsForStudies

Opis:

Zwraca liczbę wolnych miejsc na dane studia.

Funkcja korzysta z tabel studies i studiesParticipants, liczy liczbę studentów na konkretnych studiach i odejmuje ją od wartości placeLimit.

Wejście:

- @studiesID (int)

Wyjście:

- int

12. availableSeatsForCourses

Opis: Zwraca liczbę wolnych miejsc na kursie.

Spośród kursu znajduje wszystkie spotkania w danym kursie, sprawdza które spotkania są stacjonarnie. Następnie z tabeli rooms uzyskuje spotkanie, których pokój może pomieścić maksymalnie najmniejszą liczbę osób z wszystkich stacjonarnych spotkań i zapisuje tą wartość. Z tabeli CourseAccess liczy liczbę studentów, którzy mają dostęp do kursu i na koniec odejmując od tej liczby dostępnych miejsc liczbę studentów zapisanych na kurs. Zwraca tą wartość.

Wejście:

- @courseID (int)

Wyjście:

- int

```

CREATE FUNCTION availableSeatsForCourses (
    @courseID INT
)
RETURNS INT
AS
BEGIN
    DECLARE @minRoomCapacity INT;
    DECLARE @studentsWithAccess INT;
    DECLARE @availableSeats INT;

    SELECT @minRoomCapacity = MIN(r.placeLimit)
    FROM CourseModules cm
    JOIN moduleMeetings mm  1<->1..n: ON cm.moduleID = mm.moduleID
    JOIN meeting m  1<->1: ON mm.meetingID = m.meetingID
    JOIN StationaryMeeting sm  1<->1: ON m.meetingID = sm.meetingID
    JOIN Rooms r  1..n<->1: ON sm.room = r.id
    WHERE cm.courseID = @courseID;

    SELECT @studentsWithAccess = COUNT(*)
    FROM CourseAccess
    WHERE courseID = @courseID;

    SET @availableSeats = @minRoomCapacity - @studentsWithAccess;

    IF @availableSeats < 0
    BEGIN
        SET @availableSeats = 0;
    END

    RETURN @availableSeats;
END

```

13. availableSeatsForStudiesSession

Opis: Zwraca ile jest jeszcze wolnych miejsc na jednej sesji studyjnej.

Sprawdza ile jest dostępnych wszystkich miejsc w pokoju na sesji studyjnej. Następnie sprawdza ile jest już zajęty przez powiązanie ile osób jest na danym module. Ilość osób na modulu to ilość zajętych miejsc na spotkaniu z tego modułu przez studentów studiów. Do tego dolicza ilość wpisów studentów dla spotkania studyjnego spośród extraStudySessionParticipants. Od wszystkich dostępnych miejsc odejmuje liczbę osób z modułu i tych dodatkowo zapisanych. Zwraca wynik.

Wejście:

- @mettingID (int)
- @moduleID (int)
- @studiesID (int)

Wyjście:

- int

```
CREATE FUNCTION availableSeatsForStudiesSession (
    @meetingID INT,
    @moduleID INT,
    @studiesID INT
)
RETURNS INT
AS
BEGIN
    DECLARE @roomCapacity INT;
    DECLARE @studentsFromModule INT;
    DECLARE @extraParticipants INT;
    DECLARE @availableSeats INT;

    SELECT @roomCapacity = r.placeLimit
    FROM studySessions ss
    JOIN StationaryMeeting sm ON ss.meetingID = sm.meetingID
    JOIN Rooms r [1..n<->1] ON sm.room = r.id
    WHERE ss.meetingID = @meetingID AND ss.studiesID = @studiesID AND ss.moduleID = @moduleID;

    SELECT @studentsFromModule = COUNT(*)
    FROM studyModuleParticipants smp
    WHERE smp.studiesID = @studiesID AND smp.moduleID = @moduleID;

    SELECT @extraParticipants = COUNT(*)
    FROM extraStudySessionsParticipants esp
    WHERE esp.meetingID = @meetingID AND esp.studiesID = @studiesID AND esp.moduleID = @moduleID;

    SET @availableSeats = @roomCapacity - (@studentsFromModule + @extraParticipants);

    IF @availableSeats < 0
    BEGIN
        SET @availableSeats = 0;
    END

    RETURN @availableSeats;
END
```

Funkcje wspierające raportowanie i kontrolę płatności

14. meetingAttendanceReport

Opis:

Funkcja zwraca raport obecności na danym spotkaniu (meetingID) wraz z imieniem i nazwiskiem studenta oraz informacją, czy był obecny. Funkcja jest przydatna do generowania list obecności i raportowania frekwencji.

Funkcja łączy tabelę meetingAttendance z tabelami students i users, aby uzyskać szczegóły o studentach przypisanych do danego spotkania (meetingID). Funkcja wybiera dane tylko dla wierszy, gdzie meetingID odpowiada przekazanemu parametrowi.

Wejście:

- @meetingID (INT): Identyfikator spotkania.

Wyjście:

- Zestaw wierszy zawierających:
 - studentID (INT): Identyfikator studenta.
 - firstName (NVARCHAR): Imię studenta.
 - lastName (NVARCHAR): Nazwisko studenta.
 - attendance (BIT): Informacja o obecności (1 – obecny, 0 – nieobecny).

```
CREATE FUNCTION meetingAttendanceReport (
    @meetingID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        ma.studentID,
        u.firstName,
        u.lastName,
        ma.attendance
    FROM meetingAttendance ma
        JOIN students s [1..n] ON ma.studentID = s.studentID
        JOIN users u [1..1] ON s.studentID = u.userID
    WHERE ma.meetingID = @meetingID
)
go
```

15. singleProductPriceHistory

Opis:

Funkcja zwraca historię zmian ceny dla danego produktu (productID) na podstawie tabeli ProductPriceHistory. Każdy rekord w wyniku zawiera zakres dat, przez który cena obowiązywała, oraz wartość ceny. Funkcja wybiera wiersze z tabeli ProductPriceHistory, które odpowiadają podanemu @productID.

Wejście:

- @productID (INT): Identyfikator produktu.

Wyjście:

- Zestaw wierszy zawierających:
 - startDate (DATETIME): Data rozpoczęcia obowiązywania ceny.
 - endDate (DATETIME): Data zakończenia obowiązywania ceny.
 - price (MONEY): Cena produktu w tym okresie.

```
CREATE FUNCTION singleProductPriceHistory (
    @productID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        pph.startDate,
        pph.endDate,
        pph.price
    FROM ProductPriceHistory pph
    WHERE pph.productID = @productID
)
go
```

16. totalRevenueForProduct

Opis:

Oblicza całkowity przychód wygenerowany przez określony produkt (webinar/kurs/studia), bazując na orders, orderDetails i products.

Funkcja łączy orderDetails z orders i products poprzez productID. Następnie oblicza całkowitą ilość zamówień tego produktu w tabeli orderDetails i mnoży przez cenę produktu.

Wejście:

- @productID (int)

Wyjście:

- money

```
CREATE FUNCTION totalRevenueForProduct (
    @productID INT
)
RETURNS MONEY
AS
BEGIN
    DECLARE @totalRevenue MONEY = 0;

    SELECT @totalRevenue = COUNT(od.productID) * MAX(p.price)
    FROM orderDetails od
        JOIN products p 1..n<->1: ON od.productID = p.productID
    WHERE od.productID = @productID;

    RETURN @totalRevenue;
END
go
```

Funkcje wspierające analizę wydarzeń

17. eventsOverlap

Opis:

Sprawdza, czy dwa wskazane wydarzenia (spotkania studyjne, spotkania kursów, webinary) nachodzą na siebie czasowo.

Funkcja wyszukuje meeting po meetingID (jeśli event to pojedyncze spotkanie) lub w przypadku webinary wyszukuje ramy czasowe webinaru (najwcześniejszy startDate i najpóźniejszy endDate). Porównuje zakres czasowy dwóch wydarzeń.

Wejście:

- @eventId1 (int)
- @eventId2 (int)

Wyjście:

- bit (1 – nachodzą, 0 – nie nachodzą)

```

CREATE FUNCTION eventsOverlap (
    @eventID1 INT,
    @eventID2 INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @overlap BIT = 0;
    DECLARE @startDate1 DATETIME, @endDate1 DATETIME;
    DECLARE @startDate2 DATETIME, @endDate2 DATETIME;

    SELECT TOP 1
        @startDate1 = CASE
            WHEN meetingID IS NOT NULL THEN m.startDate
            ELSE w.startDate
        END,
        @endDate1 = CASE
            WHEN meetingID IS NOT NULL THEN m.endDate
            ELSE w.endDate
        END
    FROM meeting m
    LEFT JOIN Webinars w ON m.meetingID = @eventID1
    WHERE m.meetingID = @eventID1 OR w.webinarID = @eventID1;

    SELECT TOP 1
        @startDate2 = CASE
            WHEN m.meetingID IS NOT NULL THEN m.startDate
            ELSE w.startDate
        END,
        @endDate2 = CASE
            WHEN meetingID IS NOT NULL THEN m.endDate
            ELSE w.endDate
        END
    FROM meeting m
    LEFT JOIN Webinars w ON m.meetingID = @eventID2
    WHERE m.meetingID = @eventID2 OR w.webinarID = @eventID2;

    IF (@startDate1 <= @endDate2 AND @endDate1 >= @startDate2)

```

```
BEGIN
    SET @overlap = 1;
END

RETURN @overlap;
END
go
```

Funkcje uzupełniające logikę języków i tłumaczeń

18. isTranslatorAssignedToMeeting

Opis:

Sprawdza, czy do danego spotkania jest przypisany tłumacz i zwraca ID języka tłumaczenia. Funkcja sprawdza kolumnę meetingTranslator w tabeli meeting. Jeśli translatorID jest przypisany, znajduje languageID w tabeli translators, a następnie zwraca nazwę języka z tabeli languages.

Wejście:

- @meetingID (int)

Wyjście:

- int (ID języka) lub NULL

```

CREATE FUNCTION isTranslatorAssignedToMeeting (
    @meetingID INT
)
RETURNS NVARCHAR(64)
AS
BEGIN
    DECLARE @languageName NVARCHAR(64);

    SELECT @languageName = CASE
        WHEN m.meetingTranslator IS NOT NULL THEN l.name
        ELSE NULL
    END
    FROM meeting m
    LEFT JOIN translators t [1..n<->0..1] ON m.meetingTranslator = t.translatorID
    LEFT JOIN languages l [1..n<->1] ON t.languageID = l.id
    WHERE m.meetingID = @meetingID;

    RETURN @languageName;
END
go

grant execute on dbo.isTranslatorAssignedToMeeting to Participant
go

```

Funkcje wspierające zgodność z regulacjami

19. userRODOCompleted

Opis:

Sprawdza, czy użytkownik ma wypełnione zgody RODO.

Funkcja szuka w tabeli RODO rekordów z userID = @studentID i consent = 1. Jeśli taki rekord istnieje, zwraca 1, w przeciwnym razie 0.

Wejście:

- @studentID (int)

Wyjście:

- bit (1 – RODO wypełnione, 0 – brak zgody)

Funkcje wspierające harmonogramy

20. GetStudySchedule

Opis:

Funkcja zwraca harmonogram zajęć (spotkań) dla danego kierunku studiów o identyfikatorze @StudiesID.

Wybiera wszystkie sesje przypisane do wskazanego kierunku (studySessions), a następnie pobiera szczegóły spotkań z tabeli meetings, takie jak startDate i endDate.

Wejście:

- @StudiesID (int): identyfikator kierunku studiów

Wyjście:

- Zestaw wierszy zawierających:
 - MeetingID,
 - MeetingName,
 - startDate,
 - endDate.

```
CREATE FUNCTION GetStudySchedule (
    @StudiesID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        m.meetingID,
        m.startDate,
        m.endDate
    FROM studyModules sm
    JOIN studySessions ss [ 1<->1..n: ] ON sm.studiesID = ss.studiesID AND sm.moduleID = ss.moduleID
    JOIN meeting m [ 1..n<->1: ] ON ss.meetingID = m.meetingID
    WHERE sm.studiesID = @StudiesID
)
go
```

21. GetStudyModulesSchedule

Opis:

Funkcja zwraca harmonogram zajęć (spotkań) dla danego modułu kierunku studiów o identyfikatorze @ModuleID i @StudiesID.

Wybiera sesje z tabeli studySessions powiązane z modułem i kierunkiem studiów, a następnie łączy je z meetings, tutors i users, aby zwrócić szczegóły spotkań oraz dane prowadzących.

Wejście:

- @StudiesID (int): identyfikator kierunku studiów

- @ModuleID (int): identyfikator modułu

Wyjście:

- Zestaw wierszy zawierających:
 - MeetingID,
 - MeetingName,
 - startDate,
 - endDate,
 - tutorFirstName,
 - tutorLastName.

```

CREATE FUNCTION GetStudyModulesSchedule (
    @StudiesID INT,
    @ModuleID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        m.meetingID,
        m.startDate,
        m.endDate,
        u.firstName AS tutorFirstName,
        u.lastName AS tutorLastName
    FROM studySessions ss
    JOIN meeting m [1..n<->1] ON ss.meetingID = m.meetingID
    JOIN tutors t [1..n<->1] ON m.tutorID = t.tutorID
    JOIN users u [1<->1] ON t.tutorID = u.userID
    WHERE ss.studiesID = @StudiesID AND ss.moduleID = @ModuleID
)
go

```

22. GetCourseSchedule

Opis:

Funkcja zwraca harmonogram zajęć (spotkań) dla danego kursu @CourseID.

Wybiera moduły kursu z tabeli CourseModules, a następnie przez moduleMeetings i meetings pobiera szczegóły spotkań (startDate, endDate, meetingName).

Wejście:

- @CourseID (int): identyfikator kursu

Wyjście:

- Zestaw wierszy zawierających:
 - MeetingID,
 - MeetingName,
 - startDate,
 - endDate.

```
CREATE FUNCTION GetCourseSchedule (
    @CourseID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        m.meetingID,
        m.startDate,
        m.endDate
    FROM CourseModules cm
        JOIN moduleMeetings mm 1<->1..n: ON cm.moduleID = mm.moduleID
        JOIN meeting m 1<->1: ON mm.meetingID = m.meetingID
    WHERE cm.courseID = @CourseID
)
```

23. getCourseModulesSchedule

Opis:

Funkcja zwraca harmonogram zajęć (spotkań) dla danego modułu kursu o identyfikatorze @ModuleID i @CourseID.

Wybiera moduły kursu z tabeli CourseModules, a następnie przez moduleMeetings i meetings pobiera szczegóły spotkań, takie jak: meetingID, startDate, endDate, meetingName. Łączy meetings z tutors i users, aby uzyskać dane prowadzących.

Wejście:

- @CourseID (int): identyfikator kursu
- @ModuleID (int): identyfikator modułu

Wyjście:

- Zestaw wierszy zawierających:
 - MeetingID,
 - MeetingName,
 - startDate,
 - endDate,
 - tutorFirstName,
 - tutorLastName.

```

CREATE FUNCTION getCourseModulesSchedule (
    @CourseID INT,
    @ModuleID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        m.meetingID,
        m.startDate,
        m.endDate,
        u.firstName AS tutorFirstName,
        u.lastName AS tutorLastName
    FROM CourseModules cm
        JOIN moduleMeetings mm 1<->1..n: ON cm.moduleID = mm.moduleID
        JOIN meeting m 1<->1: ON mm.meetingID = m.meetingID
        JOIN tutors t 1..n<->1: ON m.tutorID = t.tutorID
        JOIN users u 1<->1: ON t.tutorID = u.userID
    WHERE cm.courseID = @CourseID AND cm.moduleID = @ModuleID
)

```

24. GetTimeTableForStudent

Opis:

Funkcja zwraca pełny harmonogram wszystkich spotkań dla danego studenta @StudentID z uwzględnieniem wszystkich form kształcenia: studiów, kursów i webinarów.

Łączy dane z tabel studiesParticipants, studySessions, meetings, tutors i users dla studiów;

CourseAccess, CourseModules, moduleMeetings, meetings, tutors i users dla kursów; oraz WebinarAccess, Webinars, meetings dla webinarów. Wyniki z trzech sekcji są łączone za pomocą UNION ALL.

Wejście:

- @StudentID (int): identyfikator studenta

Wyjście:

- Zestaw wierszy zawierających:
 - eventType (typ wydarzenia: Study Meeting, Course Module, Webinar),
 - MeetingID,
 - MeetingName,
 - startDate,
 - endDate,
 - tutorFirstName (dla kursów i studiów),
 - tutorLastName (dla kursów i studiów).

```

CREATE FUNCTION GetTimeTableForStudent (
    @StudentID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT
        'Study Meeting' AS eventType,
        m.meetingID,
        m.startDate,
        m.endDate,
        u.firstName AS tutorFirstName,
        u.lastName AS tutorLastName
    FROM studiesParticipants sp
    JOIN studySessions ss ON sp.studiesID = ss.studiesID
    JOIN meeting m [1..n<->1] ON ss.meetingID = m.meetingID
    JOIN tutors t [1..n<->1] ON m.tutorID = t.tutorID
    JOIN users u [1<->1] ON t.tutorID = u.userID
    WHERE sp.studentID = @StudentID

    UNION ALL

    SELECT
        'Course Module' AS eventType,
        meetingID m.meetingID,
        startDate m.startDate,
        endDate m.endDate,
        u.firstName AS tutorFirstName,
        u.lastName AS tutorLastName
    FROM CourseAccess ca
    JOIN CourseModules cm ON ca.courseID = cm.courseID
    JOIN moduleMeetings mm [1<->1..n] ON cm.moduleID = mm.moduleID
    JOIN meeting m [1<->1] ON mm.meetingID = m.meetingID
    JOIN tutors t [1..n<->1] ON m.tutorID = t.tutorID
    JOIN users u [1<->1] ON t.tutorID = u.userID
    WHERE ca.studentID = @StudentID

    UNION ALL
)

```

```
SELECT
    'Webinar' AS eventType,
    meetingID m.meetingID,
    startDate m.startDate,
    endDate m.endDate,
    NULL AS tutorFirstName,
    NULL AS tutorLastName
FROM WebinarAccess wa
JOIN Webinars w 1..n<->1: ON wa.webinarID = w.webinarID
JOIN meeting m ON w.webinarID = m.meetingID
WHERE wa.studentID = @StudentID
)
go
```

25. isStudentPresent

Opis:

Funkcja sprawdza, czy student o podanym @student_id był obecny na spotkaniu o @meeting_id lub spełnił warunki zaliczenia tego spotkania (np. w przypadku zajęć asynchronicznych).

Najpierw sprawdza obecność w tabeli meetingAttendance. Jeśli brak obecności, sprawdza w moduleMeetings i courseModulesPassed, czy student zaliczył moduł powiązany z tym spotkaniem.

Wejście:

- @student_id (int): identyfikator studenta
- @meeting_id (int): identyfikator spotkania

Wyjście:

- bit (1 – obecny/zaliczone, 0 – brak obecności/zaliczenia)

```
CREATE FUNCTION isStudentPresent (
    @student_id INT,
    @meeting_id INT
)
RETURNS BIT
AS
BEGIN
    DECLARE @isPresent BIT = 0;

    IF EXISTS (
        SELECT 1
        FROM meetingAttendance ma
        WHERE ma.studentID = @student_id
            AND ma.meetingID = @meeting_id
            AND ma.attendance = 1
    )
    BEGIN
        SET @isPresent = 1;
    END
    ELSE
    BEGIN
        IF EXISTS (
            SELECT 1
            FROM moduleMeetings mm
            JOIN courseModulesPassed cmp ON mm.moduleID = cmp.moduleID
            WHERE mm.meetingID = @meeting_id
                AND cmp.studentID = @student_id
        )
        BEGIN
            SET @isPresent = 1;
        END
    END
END

RETURN @isPresent;
END
go
```

26. getShoppingCartValue

Opis:

Funkcja oblicza całkowitą wartość koszyka studenta o identyfikatorze @student_id. Łączy dane z tabel cart, cartDetails i products, aby uzyskać ceny produktów znajdujących się w koszyku studenta. Następnie sumuje ceny produktów. Jeśli koszyk jest pusty, zwraca 0.

Wejście:

- @student_id (int): identyfikator studenta

Wyjście:

- MONEY: całkowita wartość koszyka

```
CREATE FUNCTION getShoppingCartValue (
    @student_id INT
)
RETURNS MONEY
AS
BEGIN
    DECLARE @totalValue MONEY = 0;

    SELECT @totalValue = SUM(p.price)
    FROM cart c
        JOIN cartDetails cd  1<->1..n: ON c.cartID = cd.cartID
        JOIN products p   1..n<->1: ON cd.productID = p.productID
    WHERE c.studentID = @student_id;

    RETURN ISNULL(@totalValue, 0);
END
go

grant execute on dbo.getShoppingCartValue to Participant
go
```

PROCEDURY

Sekcja Users

userAdd

Dodanie nowego użytkownika przez podanie odpowiednich danych i zwrócenie User_id. Argumenty procedury:

- @firstName - imię użytkownika
- @lastName – nazwisko użytkownika
- @email – podany e-mail
- @password – podane hasło
- @userID – zwracane userID

```
CREATE PROCEDURE userAdd
    @firstName NVARCHAR(64),
    @lastName NVARCHAR(64),
    @email NVARCHAR(64),
    @password NVARCHAR(64),
    @userID INT OUTPUT

AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO users (firstName, lastName, email, password)
    VALUES ( firstName @firstName, lastName @lastName, email @email, password @password);

    SET @userID = SCOPE_IDENTITY();
END;
```

userModifyData

Modyfikacja danych użytkownika. Argumenty procedury:

- @userID – ID użytkownika
- @firstName - imię użytkownika
- @lastname – nazwisko użytkownika
- @email – podany e-mail
- @password – podane hasło

```
CREATE PROCEDURE userModifyData
@userID INT,
@firstName NVARCHAR(64),
@lastName NVARCHAR(64),
@email NVARCHAR(64),
@password NVARCHAR(64)

AS
BEGIN
    SET NOCOUNT ON;
    IF EXISTS (SELECT 1 FROM users WHERE userID = @userID)
    BEGIN

        UPDATE users
        SET
            firstName = @firstName,
            lastName = @lastName,
            email = @email,
            password = @password
        WHERE userID = @userID;

        PRINT 'Dane użytkownika zostały zaktualizowane.';

    END
    ELSE
    BEGIN

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

userDelete

Usunięcie użytkownika z bazy. Argumenty procedury:

- @userID – ID użytkownika

```
CREATE PROCEDURE userDelete
    @userID INT
AS
BEGIN
    SET NOCOUNT ON;

    -- Sprawdzenie, czy użytkownik istnieje
    IF EXISTS (SELECT 1 FROM users WHERE userID = @userID)
        BEGIN

            DELETE FROM users
            WHERE userID = @userID;

            PRINT 'Użytkownik został usunięty.';

        END
    ELSE
        BEGIN

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

        END
    END;
```

studentModifyAddress

Modyfikacja adresu studenta. Argumenty procedury:

- @studentID – ID studenta
- @country - państwo z którego pochodzi student
- @city - miasto
- @street - ulica
- @zipcode – numer pocztowy

```

CREATE PROCEDURE studentModifyAddress
    @studentID INT,
    @country NVARCHAR(64),
    @city NVARCHAR(64),
    @street NVARCHAR(64),
    @zipCode NVARCHAR(8)
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM address WHERE userID = @studentID)
    BEGIN

        UPDATE address
        SET
            country = @country,
            city = @city,
            street = @street,
            zipCode = @zipCode
        WHERE userID = @studentID;

        PRINT 'Adres został zaktualizowany.';

    END
    ELSE
    BEGIN
        RAISERROR ('Nie znaleziono adresu dla podanego studentID.', 16, 1);
    END
END;

```

addUserWithTutor

Dodanie nowego Usera do bazy i automatycznie do tabeli tutors.

Argumenty procedury:

- @firstName - imię tutora
- @lastname – nazwisko tutora
- @email – podany e-mail
- @password – podane hasło
- @userID – zwracane userID

```
CREATE PROCEDURE AddUserWithTutor
    @FirstName NVARCHAR(64),
    @LastName NVARCHAR(64),
    @Email NVARCHAR(64),
    @Password NVARCHAR(64)
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        INSERT INTO users (firstName, lastName, email, password)
        VALUES (@firstName, @lastName, @email, @password);

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

        INSERT INTO tutors (tutorID)
        VALUES (@userID @NewUserID);

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO
```

addUserWithCoordinator

Dodanie nowego Usera do bazy i automatycznie do tabeli coordinators.
Argumenty procedury:

- @firstName - imię coordinatora
- @lastname – nazwisko coordinatora
- @email – podany e-mail
- @password – podane hasło
- @userID – zwracane userID

```

CREATE PROCEDURE AddUserWithCoordinator
    @FirstName NVARCHAR(64),
    @LastName NVARCHAR(64),
    @Email NVARCHAR(64),
    @Password NVARCHAR(64)
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        INSERT INTO users (firstName, lastName, email, password)
        VALUES ( @FirstName, @LastName, @Email, @Password);

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

        INSERT INTO coordinators (coordinatorID)
        VALUES ( @NewUserID);

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO

```

addUserWithTranslator

Dodanie nowego usera do bazy i automatycznie do tabeli translators.

Argumenty procedury:

- @firstName - imię translatora
- @lastname – nazwisko translatora
- @email – podany e-mail
- @password – podane hasło
- @userID – zwracane userID
- @languageID - istniejące w bazie ID języka

```
CREATE PROCEDURE AddUserWithTranslator
    @FirstName NVARCHAR(64),
    @LastName NVARCHAR(64),
    @Email NVARCHAR(64),
    @Password NVARCHAR(64),
    @LanguageID INT
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        IF NOT EXISTS (SELECT 1 FROM languages WHERE id = @LanguageID)
        BEGIN
            THROW 50001, 'Invalid LanguageID. The specified LanguageID does not exist in the languages table.';
        END;

        INSERT INTO users (firstName, lastName, email, password)
        VALUES (@FirstName, @LastName, @Email, @Password);

        DECLARE @NewUserID INT;
        SET @NewUserID = SCOPE_IDENTITY();
        INSERT INTO translators (translatorID, languageID)
        VALUES (@NewUserID, @LanguageID);
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO
```

addUserWithStudent

Dodanie nowego usera do bazy i automatycznie do tabeli users.

Argumenty procedury:

- @firstName - imię translatora
- @lastname – nazwisko translatora
- @email – podany e-mail
- @password – podane hasło
- @userID – zwracane userID

```
CREATE PROCEDURE addUserWithStudent
    @FirstName NVARCHAR(64),
    @LastName NVARCHAR(64),
    @Email NVARCHAR(64),
    @Password NVARCHAR(64),
    @studentID INT OUTPUT
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        INSERT INTO users (firstName, lastName, email, password)
        VALUES (@FirstName, @LastName, @Email, @Password);

        SET @studentID = SCOPE_IDENTITY();

        INSERT INTO students (studentID)
        VALUES (@studentID);

        COMMIT TRANSACTION;

        PRINT 'Nowy użytkownik i student zostali dodani.';
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO
```

2024-02-01 10:05:46.000 SPLE - UTF-8 - 4 pages

deleteUserWithStudent

usunięcie usera i zarówno studenta z bazy. Argumenty procedury:

- @userID – userID

```

CREATE PROCEDURE deleteUserWithStudent
    @UserID INT
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        IF NOT EXISTS (SELECT 1 FROM users WHERE userID = @UserID)
        BEGIN
            RAISERROR('Użytkownik o podanym ID nie istnieje.', 16, 1);
            RETURN;
        END;

        IF NOT EXISTS (SELECT 1 FROM students WHERE studentID = @UserID)
        BEGIN
            RAISERROR('Student o podanym ID nie istnieje.', 16, 1);
            RETURN;
        END;

        DELETE FROM students
        WHERE studentID = @UserID;

        DELETE FROM users
        WHERE userID = @UserID;

        COMMIT TRANSACTION;

        PRINT 'Użytkownik i student zostali usunięci.';
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
    END CATCH

```

DeleteTutorAndUser

usunięcie danego tutora z tabeli tutors i users. Argumenty Procedury:

- @tutorID

```
CREATE PROCEDURE DeleteTutorAndUser
    @TutorID INT
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        IF NOT EXISTS (SELECT 1 FROM tutors WHERE tutorID = @TutorID)
        BEGIN
            THROW 50002, 'Invalid TutorID. The specified TutorID does not exist in the tutors table.', 1;
        END;

        DELETE FROM tutors
        WHERE tutorID = @TutorID;

        DELETE FROM users
        WHERE userID = @TutorID;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO
```

DeleteCoordinatorAndUser

usunięcie danego koordynatora z tabeli coordinators i users. Argumenty Procedury:

- @coordinatorID

```
CREATE PROCEDURE DeleteCoordinatorAndUser
    @CoordinatorID INT
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        IF NOT EXISTS (SELECT 1 FROM coordinators WHERE coordinatorID = @CoordinatorID)
        BEGIN
            THROW 50003, 'Invalid CoordinatorID. The specified CoordinatorID does not exist in the coordinator table.'
        END;

        DELETE FROM coordinators
        WHERE coordinatorID = @CoordinatorID;

        DELETE FROM users
        WHERE userID = @CoordinatorID;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO
```

DeleteTranslatorAndUser

usuńcie danego translatora z tabeli translators i users. Argumenty Procedury:

- @translatorID

```
CREATE PROCEDURE DeleteTranslatorAndUser
    @TranslatorID INT
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        IF NOT EXISTS (SELECT 1 FROM translators WHERE translatorID = @TranslatorID)
        BEGIN
            THROW 50004, 'Invalid TranslatorID. The specified TranslatorID does not exist in the translators t
        END;

        DELETE FROM translators
        WHERE translatorID = @TranslatorID;

        DELETE FROM users
        WHERE userID = @TranslatorID;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;

        THROW;
    END CATCH
END;
GO
```

Sekcja Meetings

addMeeting

Dodanie nowego spotkania do listy i zwrócenie meetingID. Argumenty procedury:

- @tutorID – ID prowadzącego
- @startDate – data początku spotkania
- @endDate – data końca spotkania
- @languageID – ID języka w jakim jest prowadzone spotkanie
- @meetingTranslator – ID translatora
- @meetingID – zwracana wartość meetingID

```
CREATE PROCEDURE addMeeting
    @tutorID INT,
    @startDate DATETIME,
    @endDate DATETIME,
    @languageID INT,
    @meetingTranslator INT = NULL,
    @meetingID INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO meeting (tutorID, startDate, endDate, languageID, meetingTranslator)
    VALUES ( @tutorID, @startDate, @endDate, @languageID, @meetingTranslator);

    SET @meetingID = SCOPE_IDENTITY();
END;
```

meetingModifyData

Zmiana informacji o spotkaniu. Argumenty procedury:

- @meetingID – meetingID
- @tutorID – ID prowadzącego
- @startDate – data początku spotkania
- @endDate – data końca spotkania
- @languageID – ID języka w jakim jest prowadzone spotkanie
- @meetingTranslator – ID translatora
- @meetingID – zwracana wartość meetingID

```

CREATE PROCEDURE meetingModifyData
    @meetingID INT,
    @tutorID INT,
    @startDate DATETIME,
    @endDate DATETIME,
    @languageID INT,
    @meetingTranslator INT = NULL
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM meeting WHERE meetingID = @meetingID)
    BEGIN
        UPDATE meeting
        SET
            tutorID = @tutorID,
            startDate = @startDate,
            endDate = @endDate,
            languageID = @languageID,
            meetingTranslator = @meetingTranslator
        WHERE meetingID = @meetingID;

        PRINT 'Dane spotkania zostały zaktualizowane.';
    END
    ELSE
    BEGIN
        RAISERROR ('Spotkanie o podanym meetingID nie istnieje.', 16, 1);
    END
END;

```

meetingGet

Pobranie wszystkich spotkań w których uczestniczy dany student. Argumenty procedury:

- @studentID – ID studenta
- @meetingID – zwracane wartości meetingID na których student był lub nie był obecny

```
CREATE PROCEDURE meetingGet
    @studentID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM students WHERE studentID = @studentID)
    BEGIN
        RAISERROR ('Student o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;

    SELECT meetingID, attendance
    FROM meetingAttendance
    WHERE studentID = @studentID

END;
```

meetingPresence

Rejestracja obecności studenta na spotkaniu. Argumenty procedury:

- @studentID – ID studenta
- @meetingID – ID spotkania
- @attendance - informacja o obecności

```

CREATE PROCEDURE meetingPresence
    @studentID INT,
    @meetingID INT,
    @attendance BIT
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM students WHERE studentID = @studentID)
    BEGIN
        RAISERROR ('Student o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;
    IF NOT EXISTS (SELECT 1 FROM meeting WHERE meetingID = @meetingID)
    BEGIN
        RAISERROR ('Spotkanie o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;
    IF EXISTS (SELECT 1 FROM meetingAttendance WHERE studentID = @studentID AND meetingID = @meetingID)
    BEGIN
        UPDATE meetingAttendance
        SET attendance = @attendance
        WHERE studentID = @studentID AND meetingID = @meetingID;
        PRINT 'Obecność została zaktualizowana.';
    END
    ELSE
    BEGIN
        INSERT INTO meetingAttendance (studentID, meetingID, attendance)
        VALUES (@studentID, @meetingID, @attendance);

        PRINT 'Obecność została zarejestrowana.';
    END
END;

```

Sekcja Orders

productAdd

Dodanie nowego produktu do bazy danych. Argumenty procedury:

- @productName - nazwa produktu
- @price - cena produktu
- @priceUpdateDate - data modyfikacji ceny (automatycznie generuje baza)

```
CREATE PROCEDURE addProduct
    @productName NVARCHAR(64),
    @price MONEY,
    @priceUpdateDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    IF @price < 0
    BEGIN
        RAISERROR ('Cena produktu nie może być mniejsza niż 0.', 16, 1);
        RETURN;
    END;

    INSERT INTO products (productName, price, priceUpdateDate)
    VALUES (@productName, @price, GETDATE());

    PRINT 'Produkt został dodany do bazy.';
END;
```

orderCreate

Stworzenie nowego zamówienia przez studenta z produktów z jego aktualnego niepustego koszyka. Argumenty procedury:

- @orderId – ID zamówienia
- @studentID – ID studenta
- @orderDate - data zamówienia

```

CREATE PROCEDURE orderCreate
    @orderID INT OUTPUT,
    @studentID INT,
    @orderDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM students WHERE studentID = @studentID)
    BEGIN
        RAISERROR ('Student o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM cart WHERE studentID = @studentID)
    BEGIN
        RAISERROR ('Student nie ma przypisanego koszyka.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM cartDetails c
                    INNER JOIN cart ct 1..n<->1: ON c.cartID = ct.cartID
                    WHERE ct.studentID = @studentID)
    BEGIN
        RAISERROR ('Koszyk studenta jest pusty.', 16, 1);
        RETURN;
    END;

```

```

    INSERT INTO orders (studentID, orderDate)
    VALUES ( @studentID, @orderDate GETDATE());

    SET @orderID = SCOPE_IDENTITY();

    INSERT INTO orderDetails (orderID, productID)
    SELECT @orderID, c.productID
    FROM cartDetails c
    INNER JOIN cart ct 1..n<->1: ON c.cartID = ct.cartID
    WHERE ct.studentID = @studentID;

    DELETE FROM cartDetails
    WHERE cartID IN (SELECT cartID FROM cart WHERE studentID = @studentID);
    PRINT 'Zamówienie zostało utworzone.';

END;

```

productCartAdd

Dodanie produktu do koszyka. Argumenty procedury:

- @cartID – ID zamówienia
- @studentID – ID studenta
- @productID – ID dodawanego produktu

```
CREATE PROCEDURE productCartAdd
    @cartID INT,
    @studentID INT,
    @productID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM cart WHERE cartID = @cartID AND studentID = @studentID)
    BEGIN
        RAISERROR ('Koszyk o podanym ID nie istnieje dla tego studenta.', 16, 1);
        RETURN;
    END;
    IF NOT EXISTS (SELECT 1 FROM students WHERE studentID = @studentID)
    BEGIN
        RAISERROR ('Student o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;
    IF NOT EXISTS (SELECT 1 FROM products WHERE productID = @productID)
    BEGIN
        RAISERROR ('Produkt o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;

    INSERT INTO cartDetails (cartID, productID)
    VALUES (@cartID, @productID);

    PRINT 'Produkt został dodany do koszyka.';
END;
```

productModifyPrice

Zmiana ceny danego produktu. Argumenty procedury:

- @productID – ID produktu
- @price – Nowa cena produktu

```
CREATE PROCEDURE productModifyPrice
    @productID INT,
    @price MONEY
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM products WHERE productID = @productID)
    BEGIN
        RAISERROR ('Produkt o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF @price < 0
    BEGIN
        RAISERROR ('Cena produktu nie może być mniejsza niż 0.', 16, 1);
        RETURN;
    END;

    UPDATE products
    SET price = @price,
        priceUpdateDate = GETDATE()
    WHERE productID = @productID;

    PRINT 'Cena produktu została zaktualizowana.';
END;
```

ModifyPriceCurr

Zmiana kursu danej waluty. Argumenty procedury:

- @currencyID - ID waluty
- @currencyRate - nowy kurs

```

CREATE PROCEDURE ModifyPriceCurr
    @currencyID INT,
    @currencyRate FLOAT(3)
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM currencies WHERE currencyID = @currencyID)
    BEGIN
        RAISERROR ('Waluta o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF @currencyRate <= 0
    BEGIN
        RAISERROR ('Kurs waluty musi być większy niż 0.', 16, 1);
        RETURN;
    END;

    UPDATE currencies
    SET currencyRate = @currencyRate
    WHERE currencyID = @currencyID;

    PRINT 'Kurs waluty został zaktualizowany.';
END;

```

Sekcja Studies

addStudies

Dodaje nowe studia do bazy i zwraca studiesID do tych studiów. Argumenty procedury:

- @placeLimit

```
CREATE PROCEDURE addStudies
    @placeLimit INT,
    @studiesName NVARCHAR(255),
    @studiesID INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    IF @placeLimit <= 0
    BEGIN
        RAISERROR ('Limit miejsc musi być większy od 0.', 16, 1);
        RETURN;
    END;

    IF @studiesName IS NULL OR LEN(@studiesName) = 0
    BEGIN
        RAISERROR ('Nazwa studiów nie może być pusta.', 16, 1);
        RETURN;
    END;

    INSERT INTO studies (placeLimit, studiesName)
    VALUES (@placeLimit, @studiesName);

    SET @studiesID = SCOPE_IDENTITY();

    PRINT 'Nowe studia zostały dodane.';
END;
```

updateStudiesPlaceLimit

Zmienia limit miejsc na studiach. Argumenty procedury:

- @placeLimit
- @studiesID

```
CREATE PROCEDURE updateStudiesPlaceLimit
    @placeLimit INT,
    @studiesID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM studies WHERE studiesID = @studiesID)
    BEGIN
        RAISERROR ('Studia o podanym ID nie istnieja.', 16, 1);
        RETURN;
    END;

    IF @placeLimit <= 0
    BEGIN
        RAISERROR ('Limit miejsc musi być większy od 0.', 16, 1);
        RETURN;
    END;

    UPDATE studies
    SET placeLimit = @placeLimit
    WHERE studiesID = @studiesID;

    PRINT 'Limit miejsc został zaktualizowany.';
END;
```

removeStudies

Usuwa studia. Argumenty procedury:

- @studiesID

```

CREATE PROCEDURE removeStudies
    @studiesID INT
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM studies WHERE studiesID = @studiesID)
        BEGIN
            RAISERROR('Studia o podanym ID nie istnieja.', 16, 1);
            RETURN;
        END;
    BEGIN TRANSACTION;
    BEGIN TRY
        DECLARE @entryFeeProductID INT;
        SELECT @entryFeeProductID = entryFeeProductID
        FROM studies
        WHERE studiesID = @studiesID;

        DELETE FROM studies
        WHERE studiesID = @studiesID;

        IF @entryFeeProductID IS NOT NULL
            BEGIN
                DELETE FROM products
                WHERE productID = @entryFeeProductID;
            END;

        COMMIT TRANSACTION;

        PRINT 'Studia i powiązany produkt zostały usunięte.';
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
    END CATCH;

```

addInternship

Dodaje praktyki do studiów i zwraca InternshipID. Argumenty procedury:

- @studiesID
- @startDate
- @endDate

```
CREATE PROCEDURE addInternship
    @studiesID INT,
    @startDate DATE,
    @endDate DATE,
    @internshipID INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    IF @endDate <= @startDate
    BEGIN
        RAISERROR ('Data zakończenia musi być późniejsza niż data rozpoczęcia.', 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 internship (studiesID, startDate, endDate)
    VALUES (@studiesID, @startDate, @endDate);

    SET @internshipID = SCOPE_IDENTITY();

    PRINT 'Praktyki zostały dodane.';
END;
```

updateInternshipDate

Aktualizuje datę odbywania praktyk. Argumenty procedury:

- @InternshipID
- @StartDate
- @endDate

```

CREATE PROCEDURE updateInternshipDate
    @InternshipID INT,
    @StartDate DATE,
    @EndDate DATE
AS
BEGIN
    SET NOCOUNT ON;

    IF @EndDate <= @StartDate
    BEGIN
        RAISERROR ('Data zakończenia musi być późniejsza niż data rozpoczęcia.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM internship WHERE internshipID = @InternshipID)
    BEGIN
        RAISERROR ('Praktyka o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END;

    UPDATE internship
    SET startDate = @StartDate,
        endDate = @EndDate
    WHERE internshipID = @InternshipID;

    PRINT 'Daty praktyk zostały zaktualizowane.';
END;

```

addStudiesModule

Dodaje nowy moduł do studiów i zwraca id tego modułu. Argumenty procedury:

- @studiesID
- @moduleTitle
- @coordinatorID
- @description

```
CREATE PROCEDURE addStudiesModule
    @studiesID INT,
    @moduleTitle NVARCHAR(255),
    @coordinatorID INT,
    @description NVARCHAR(MAX),
    @moduleId INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM studies WHERE studiesID = @studiesID)
    BEGIN
        RAISERROR ('Studia o podanym ID nie istnieja.', 16, 1);
        RETURN;
    END;

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

    INSERT INTO studyModules (studiesID, description, moduleTitle, coordinatorID)
    VALUES (@studiesID, @description, @moduleTitle, @coordinatorID);

    SET @moduleId = SCOPE_IDENTITY();

    PRINT 'Nowy moduł został dodany.';
END;
```

updateModuleCoordinator

Zmiana koordynatora modułu. Argumenty procedury:

- @studiesID
- @moduleId
- @coordinatorID

```
CREATE PROCEDURE updateModuleCoordinator
    @studiesID INT,
    @moduleID INT,
    @coordinatorID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM studies WHERE studiesID = @studiesID)
    BEGIN
        RAISERROR ('Studia o podanym ID nie istnieja.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM studyModules WHERE moduleID = @moduleID AND studiesID = @studiesID)
    BEGIN
        RAISERROR ('Moduł o podanym ID nie istnieje w danych studiach.', 16, 1);
        RETURN;
    END;

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

    UPDATE studyModules
    SET coordinatorID = @coordinatorID
    WHERE moduleID = @moduleID AND studiesID = @studiesID;

    PRINT 'Koordynator modułu został zaktualizowany';
END.
```

updateModuleDescription

Zmiana opis modułu. Argumenty procedury:

- @studiesID
- @ModuleID
- @description

```

CREATE PROCEDURE updateModuleDescription
    @studiesID INT,          -- ID studiów
    @moduleID INT,           -- ID modułu
    @description NVARCHAR(MAX) -- Nowy opis modułu
AS
BEGIN
    SET NOCOUNT ON;

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

    IF NOT EXISTS (SELECT 1 FROM studyModules WHERE moduleID = @moduleID AND studiesID = @studiesID)
    BEGIN
        RAISERROR ('Moduł o podanym ID nie istnieje w ramach tych studiów.', 16, 1);
        RETURN;
    END;

    UPDATE studyModules
    SET description = @description
    WHERE moduleID = @moduleID AND studiesID = @studiesID;

    PRINT 'Opis modułu został zaktualizowany.';
END;

```

addStudySessions

Dodaje spotkanie do modułu studiów. Argumenty procedury:

- @studiesID
- @moduleID
- @productID
- @meetingID

```

CREATE PROCEDURE addStudySessions
    @meetingID INT,
    @studiesID INT,
    @moduleId INT,
    @productID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM StudySessions WHERE MeetingID = @meetingID)
    BEGIN
        RAISERROR ('Podany meetingID już istnieje w tabeli StudySessions.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM Products WHERE ProductID = @productID)
    BEGIN
        RAISERROR ('Podany productID nie istnieje w tabeli Products.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM Studies WHERE StudiesID = @studiesID)
    BEGIN
        RAISERROR ('Podany studiesID nie istnieje w tabeli Studies.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM StudyModules WHERE ModuleID = @moduleId AND StudiesID = @studiesID)
    BEGIN
        RAISERROR ('Podany moduleId nie istnieje dla podanego studiesID w tabeli StudyModules.', 16, 1);
        RETURN;
    END

```

```

IF NOT EXISTS (SELECT 1 FROM StudyModules WHERE ModuleID = @moduleId AND StudiesID = @studiesID)
BEGIN
    RAISERROR ('Podany moduleId nie istnieje dla podanego studiesID w tabeli StudyModules.', 16, 1);
    RETURN;
END

INSERT INTO StudySessions (MeetingID, StudiesID, ModuleID, ProductID)
VALUES ( @meetingID, @studiesID, @moduleId, @productID);

PRINT 'Spotkanie zostało pomyślnie dodane.';
END;

```

addStudentToStudies

Dodaje studenta do studiów. Argumenty procedury: @studentID, @studiesID

```
CREATE PROCEDURE addStudentToStudies
    @studentID INT,
    @studiesID INT
AS
BEGIN
    SET NOCOUNT ON;

    -- Sprawdzenie, czy student już jest zapisany na te studia
    IF EXISTS (
        SELECT 1
        FROM studiesParticipants
        WHERE studentID = @studentID AND studiesID = @studiesID
    )
    BEGIN
        PRINT 'Student już jest zapisany na te studia.';
        RETURN;
    END

    -- Sprawdzenie, czy są wolne miejsca na studiach
    IF dbo.availableSeatsForStudies( @studiesID ) <= 0
    BEGIN
        PRINT 'Brak dostępnych miejsc na studiach.';
        RETURN;
    END

    -- Dodanie studenta do studiów
    INSERT INTO studiesParticipants (studiesID, studentID)
    VALUES ( @studiesID, @studentID);

    PRINT 'Student został zapisany na studia.';
END;
```

Sekcja Webinars

addWebinar

Dodaje nowy webinar i zwraca jego id (webinarID). Argumenty procedury:

- @tutorID

- @translatorID
- @Title
- @productID
- @StartDate
- @Description
- @recording URL
- @languageID

```
CREATE PROCEDURE addWebinar
    @tutorID INT,
    @translatorID INT,
    @Title NVARCHAR(64),
    @productID INT,
    @StartDate DATETIME,
    @endDate DATETIME,
    @Description NVARCHAR(MAX),
    @recordingURL NVARCHAR(128),
    @languageID INT,
    @webinarID INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM products WHERE productID = @productID)
    BEGIN
        RAISERROR('Produkt o podanym productID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF @StartDate >= @endDate
    BEGIN
        RAISERROR('Data zakończenia webinaru musi być późniejsza niż data rozpoczęcia.', 16, 1);
        RETURN;
    END;

    INSERT INTO Webinars (tutorID, translatorID, Title, productID, StartDate, endDate, Description, recordingU
    VALUES ( tutorID @tutorID, translatorID @translatorID, Title @Title, productID @productID, StartDate @StartDate,
    SET @webinarID = SCOPE_IDENTITY();
```

removeWebinar

Usuwa istniejący Webinar. Argumenty procedury:

- @webinarID

```
CREATE PROCEDURE removeWebinar
    @webinarID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Webinars WHERE webinarID = @webinarID)
    BEGIN
        RAISERROR ('Webinar o podanym webinarID nie istnieje.', 16, 1);
        RETURN;
    END;

    DELETE FROM Webinars
    WHERE webinarID = @webinarID;

    PRINT 'Webinar został usunięty.';
END;
```

giveAccessToWebinar

Dodaje studenta do tabelki z dostepem do Webinaru (WebinarAccess). Zwraca accessID. Argumenty procedury:

- @studentID
- @webinarID
- @accessStartDate
- @accessEndDate

```
CREATE PROCEDURE giveAccessToWebinar
    @studentID INT,
    @webinarID INT,
    @accessStartDate DATETIME,
    @accessEndDate DATETIME,
    @accessID INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Students WHERE studentID = @studentID)
    BEGIN
        RAISERROR ('Student o podanym studentID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM Webinars WHERE webinarID = @webinarID)
    BEGIN
        RAISERROR ('Webinar o podanym webinarID nie istnieje.', 16, 1);
        RETURN;
    END;

    INSERT INTO WebinarAccess (studentID, webinarID, accessStartDate, accessEndDate)
    VALUES (@studentID, @webinarID, @accessStartDate, @accessEndDate);

    SET @accessID = SCOPE_IDENTITY();

    PRINT 'Dostęp do webinaru został przyznany.';
END;
```

revokeAccessToWebinar

Zabiera dostęp do webinaru studentowi. Argumenty procedury:

- @accessID

```
CREATE PROCEDURE revokeAccessToWebinar
    @accessID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM WebinarAccess WHERE accessID = @accessID)
    BEGIN
        RAISERROR ('Dostęp o podanym accessID nie istnieje.', 16, 1);
        RETURN;
    END;

    DELETE FROM WebinarAccess
    WHERE accessID = @accessID;

    PRINT 'Dostęp do webinaru został odebrany.';
END;
```

Sekcja Courses

addCourse

Dodaje nowy kurs i zwraca CourseID. Argumenty procedury:

- @coordinatorID
- @title
- @description
- @productID
- @startDate

```
CREATE PROCEDURE addCourse
    @coordinatorID INT,
    @title NVARCHAR(100),
    @description NVARCHAR(MAX),
    @productID INT,
    @startDate DATE,
    @courseID INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Coordinators WHERE coordinatorID = @coordinatorID)
    BEGIN
        RAISERROR ('Koordynator o podanym coordinatorID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM Products WHERE productID = @productID)
    BEGIN
        RAISERROR ('Produkt o podanym productID nie istnieje.', 16, 1);
        RETURN;
    END;

    INSERT INTO Courses (coordinatorID, title, description, productID, startDate)
    VALUES (@coordinatorID, @title, @description, @productID, @startDate);

    SET @courseID = SCOPE_IDENTITY();

    PRINT 'Kurs został dodany do bazy.';
END;
```

removeCourse

Usuwa kurs. Argumenty procedury:

- @courseID

```
CREATE PROCEDURE removeCourse
    @courseID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Courses WHERE courseID = @courseID)
    BEGIN
        RAISERROR ('Kurs o podanym courseID nie istnieje.', 16, 1);
        RETURN;
    END;

    DELETE FROM Courses
    WHERE courseID = @courseID;

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

addCourseModule

Dodaje moduł do kursu. Argumenty procedury:

- @CourseID
- @tutorID
- @moduleTitle
- @moduleDate
- @languageID
- @moduleId

```

CREATE PROCEDURE addCourseModule
    @CourseID INT,
    @tutorID INT,
    @moduleID INT,
    @moduleTitle NVARCHAR(100),
    @moduleDate DATE,
    @languageID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Courses WHERE CourseID = @CourseID)
    BEGIN
        RAISERROR('Nie istnieje kurs o podanym CourseID.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM Tutors WHERE TutorID = @tutorID)
    BEGIN
        RAISERROR('Nie istnieje tutor o podanym tutorID.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM Languages WHERE id = @languageID)
    BEGIN
        RAISERROR('Nie istnieje język o podanym languageID.', 16, 1);
        RETURN;
    END

```

```

IF NOT EXISTS (SELECT 1 FROM Languages WHERE id = @languageID)
BEGIN
    RAISERROR('Nie istnieje język o podanym languageID.', 16, 1);
    RETURN;
END

IF EXISTS (SELECT 1 FROM CourseModules WHERE ModuleID = @moduleID)
BEGIN
    RAISERROR('Podany moduleID już istnieje w tabeli CourseModules.', 16, 1);
    RETURN;
END

INSERT INTO CourseModules (CourseID, TutorID, ModuleID, ModuleTitle, ModuleDate, LanguageID)
VALUES (@courseID, @tutorID, @moduleID, @moduleTitle, @moduleDate, @languageID)

PRINT 'Moduł został pomyślnie dodany.';
END;

```

removeCourseModule

Usuwa moduł z kursu. Argumenty procedury:

- @CourseID
- @ModuleID

```
CREATE PROCEDURE removeCourseModule
    @CourseID INT,
    @ModuleID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (
        SELECT 1
        FROM CourseModules
        WHERE CourseID = @CourseID AND ModuleID = @ModuleID
    )
    BEGIN
        RAISERROR('Moduł o podanym ModuleID nie istnieje w ramach wskazanego kursu.', 16, 1);
        RETURN;
    END;

    DELETE FROM CourseModules
    WHERE CourseID = @CourseID AND ModuleID = @ModuleID;

    PRINT 'Moduł został pomyślnie usunięty z kursu.';
END;
```

giveAccessToCourse

Daje studentowi dostęp do części online kursu. Argumenty procedury:

- @courseID
- @studentID
- @accessEndDate
- @accessStartDate

```
CREATE PROCEDURE giveAccessToCourse
    @courseID INT,
    @studentID INT,
    @accessStartDate DATETIME,
    @accessEndDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Courses WHERE CourseID = @courseID)
    BEGIN
        RAISERROR('Kurs o podanym CourseID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM Students WHERE StudentID = @studentID)
    BEGIN
        RAISERROR('Student o podanym StudentID nie istnieje.', 16, 1);
        RETURN;
    END;

    INSERT INTO CourseAccess (CourseID, StudentID, AccessStartDate, AccessEndDate)
    VALUES (@courseID, @studentID, @accessStartDate, @accessEndDate);

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

revokeAccessToCourse

Odbiera studentowi dostęp do kursu. Argumenty procedury:

- @courseID
- @studentID

```

CREATE PROCEDURE revokeAccessToCourse
    @courseID INT,
    @studentID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM Courses WHERE CourseID = @courseID)
    BEGIN
        RAISERROR('Kurs o podanym CourseID nie istnieje.', 16, 1);
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM Students WHERE StudentID = @studentID)
    BEGIN
        RAISERROR('Student o podanym StudentID nie istnieje.', 16, 1);
        RETURN;
    END;

    DELETE FROM CourseAccess
    WHERE CourseID = @courseID AND StudentID = @studentID;

    PRINT 'Dostęp do kursu został usunięty dla danego studenta.';
END;

```

AddStudentToCourseModulesPassed

Dodaje studenta do tabeli z zaliczonymi modułami. Argumenty Procedury:

- @moduleID - ID zaliczonego modułu
- @studentID - ID studenta, który zaliczył ten moduł

```

CREATE PROCEDURE AddStudentToCourseModulesPassed
    @moduleId INT,
    @studentID INT
AS
BEGIN
    SET NOCOUNT ON;

    IF NOT EXISTS (SELECT 1 FROM students WHERE StudentID = @studentID)
    BEGIN
        RAISERROR ('Student o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM CourseModules WHERE ModuleID = @moduleId)
    BEGIN
        RAISERROR ('Moduł o podanym ID nie istnieje.', 16, 1);
        RETURN;
    END

    IF EXISTS (SELECT 1 FROM CourseModulesPassed WHERE ModuleID = @moduleId AND StudentID = @studentID)
    BEGIN
        RAISERROR ('Student już zaliczył ten moduł.', 16, 1);
        RETURN;
    END

    INSERT INTO CourseModulesPassed (ModuleID, StudentID)
    VALUES (@moduleId, @studentID);

    PRINT 'Student został pomyślnie dodany do tabeli zaliczonych modułów.';
END;

```

TRIGGERY

Automatyczne przydzielanie dostępu do kursu studentowi przy jego zakupie

```

CREATE TRIGGER trg_AddStudentToCourse
ON dbo.OrderPayment
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @orderID INT;
    DECLARE @studentID INT;
    DECLARE @courseID INT;
    DECLARE @paymentStatus BIT;
    DECLARE @accessStartDate DATETIME;
    DECLARE @accessEndDate DATETIME;
    SET @accessStartDate = GETDATE();
    SET @accessEndDate = DATEADD (DAY, 365, @accessStartDate);
    SELECT
        @orderID = o.orderID,
        @studentID = o.studentID,
        @paymentStatus = i.PaymentStatus
    FROM
        inserted i
    JOIN
        orders o ON i.orderID = o.orderID;
    IF @paymentStatus = 1
    BEGIN
        SELECT @courseID = od.productID
        FROM orderDetails od
        WHERE od.orderID = @orderID
        AND EXISTS (
            SELECT 1
            FROM Courses c
            WHERE c.productID = od.productID
        );
        IF dbo.availableSeatsForCourses( @courseID ) > 0
        BEGIN
            EXEC dbo.[giveAccessToCourse]
                @courseID = @courseID,
                @studentID = @studentID,
                @accessStartDate = @accessStartDate,
                @accessEndDate = @accessEndDate;
        END
    END

```

Automatyczne przydzielanie dostępu do Webinaru studentowi przy jego zakupie

```
CREATE TRIGGER trg_AddStudentToWebinar
ON OrderPayment
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @studentID INT, @webinarID INT, @startDate DATETIME, @endDate DATETIME;

    DECLARE cur CURSOR FOR
        SELECT
            o.studentID,
            w.webinarID,
            w.startDate AS accessStartDate,
            DATEADD(DAY, 30, w.startDate) AS accessEndDate
        FROM inserted i
        JOIN orders o ON i.orderID = o.orderID
        JOIN orderDetails od 1<->1..n: ON o.orderID = od.orderID
        JOIN Webinars w ON od.productID = w.productID
        WHERE i.PaymentStatus = 1; -- Płatność zatwierdzona

    OPEN cur;

    FETCH NEXT FROM cur INTO @studentID, @webinarID, @startDate, @endDate;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        EXEC giveAccessToWebinar
            @studentID = @studentID,
            @webinarID = @webinarID,
            @accessStartDate = @startDate,
            @accessEndDate = @endDate;

        FETCH NEXT FROM cur INTO @studentID, @webinarID, @startDate, @endDate;
    END

    CLOSE cur;
    DEALLOCATE cur;
END;
```

Automatyczne dodanie użytkownika do spotkań studyjnych przy jego zakupie

```
CREATE TRIGGER trg_AddStudentToStudySession
ON dbo.OrderPayment
FOR UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (
        SELECT 1
        FROM Inserted i
        JOIN Deleted d ON i.paymentID = d.paymentID
        WHERE d.PaymentStatus = 0 AND i.PaymentStatus = 1
    )
    BEGIN
        DECLARE @studentID INT;
        DECLARE @productID INT;
        DECLARE @meetingID INT;
        DECLARE @studiesID INT;
        DECLARE @moduleID INT;

        SELECT @studentID = o.studentID,
               @productID = od.productID
        FROM Inserted i
        JOIN orders o ON i.orderID = o.orderID
        JOIN orderDetails od 1<->1..n: ON o.orderID = od.orderID;

        SELECT TOP 1
            @meetingID = ss.meetingID,
            @studiesID = ss.studiesID,
            @moduleID = ss.moduleID
        FROM studySessions ss
        WHERE ss.productID = @productID;

        IF dbo.availableSeatsForStudiesSession( @meetingID @meetingID, @moduleID @moduleID, @studiesID @studiesID) > 0
        BEGIN
            INSERT INTO extraStudySessionsParticipants (studiesID, meetingID, moduleID, studentID)
            VALUES ( @studiesID @studiesID, @meetingID @meetingID, @moduleID @moduleID, @studentID @studentID);
        END
    END
END:
```

Automatycznie dodanie starej ceny do historii produktu gdy ta zostanie zaktualizowana

```
CREATE TRIGGER trg_AddOldPriceToHistory
ON products
FOR UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO ProductPriceHistory (productID, price, startDate, endDate)
    SELECT
        d.productID,
        d.price,
        startDate d.priceUpdateDate,
        GETDATE() AS endDate
    FROM Deleted d
    WHERE d.price <> (SELECT price FROM Inserted i WHERE i.productID = d.productID);

    UPDATE products
    SET priceUpdateDate = GETDATE()
    FROM Inserted i
    WHERE products.productID = i.productID;
END;
```

Automatyczne zaliczenie kursu, gdy liczba zaliczonych modułów przekroczy 80%.

```

CREATE TRIGGER trg_AddStudentToAchievements_Course
ON courseModulesPassed
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @moduleID INT;
    DECLARE @studentID INT;
    DECLARE @courseID INT;
    DECLARE @totalModules INT;
    DECLARE @passedModules INT;
    DECLARE @completionPercentage DECIMAL(5, 2);

    SELECT
        @moduleID = i.moduleID,
        @studentID = i.studentID
    FROM inserted i;

    SELECT @courseID = cm.courseID
    FROM CourseModules cm
    WHERE cm.moduleID = @moduleID;

    SELECT @totalModules = COUNT(*)
    FROM CourseModules
    WHERE courseID = @courseID;

    SELECT @passedModules = COUNT(*)
    FROM courseModulesPassed cmp
    JOIN CourseModules cm 1..n<->1: ON cmp.moduleID = cm.moduleID
    WHERE cm.courseID = @courseID AND cmp.studentID = @studentID;
    SET @completionPercentage = (@passedModules * 1.0 / @totalModules) * 100;

    IF @completionPercentage >= 80
    BEGIN

```

```

        IF NOT EXISTS (
            SELECT 1
            FROM StudentAchievements
            WHERE studentID = @studentID AND achievementType = 'Course' AND achievementIDValue = @courseID
        )
        BEGIN
            INSERT INTO StudentAchievements (studentID, achievementType, achievementIDValue, certificateSent)
            VALUES (@studentID, 'Course', @courseID, 0);
        END
    END;
END;

```

INDEKSY

1. Unikalny indeks na kolumnie e-mail w tabeli users
-

Do szybkiego wyszukiwania użytkowników po e-mailu

```
CREATE UNIQUE INDEX idx_unique_email ON users(email);
```

2. Indeks na tabeli address dla country
-

do lepszego wyszukiwania użytkowników w bazie po kraju

```
CREATE INDEX idx_country_address  
ON address(country)
```

3. Indeks na tabeli orders dla studentID
-

Przydatne do wyszukiwania zamówień dla konkretnego studenta

```
CREATE INDEX idx_studentID_orders ON orders(studentID);
```

4. Indeks złożony na meetingID i attendance w tabeli meetingAttendance
-

Przyspiesza wyszukiwanie frekwencji na konkretnym spotkaniu

```
CREATE INDEX idx_meetingAttendance_meetingID_attendance  
ON meetingAttendance (meetingID, attendance);
```

5. Indeks na tabeli products dla productName

przydatne przy wyszukiwaniu produktów po nazwie

```
CREATE INDEX idx_productName_products  
ON products(productName);
```

6. Indeks na tabeli products dla price

Potrzebne przy częstym wyszukiwaniu produktów po cenie

```
CREATE INDEX idx_price_products  
ON products(price);
```

7. Indeks na tabeli studies dla placeLimit

Do szybszego wyszukiwania po limicie miejsc na studia

```
CREATE INDEX idx_placeLimit_studies  
ON studies(placeLimit)
```

8. Indeks na tabeli Rooms dla placeLimit

Do szybszego wyszukiwania po limicie miejsc w salach

```
CREATE INDEX idx_placeLimit_Rooms  
ON Rooms(placeLimit)
```

9. Indeks na tabeli currencies dla currencyName

Do efektywniejszego wyszukiwania walut po nazwie

```
CREATE INDEX idx_currencyName_currencies  
ON currencies (currencyName);
```

10. Indeks na tabeli ProductPriceHistory dla price

Do szybszego wyszukiwania określonych cen dla produktów

```
CREATE INDEX inx_price_ProductPriceHistory  
ON ProductPriceHistory(price)
```

11. Indeks na tabeli internship dla startDate

Do lepszego wyszukiwania po dacie startu praktyk

```
CREATE INDEX idx_startDate_internship  
ON internship (startDate);
```

12. Indeks na tabeli CourseModules dla moduleDate

Szybsze wyszukiwanie po dacie kiedy odbywa się dany moduł

```
CREATE INDEX idx_moduleDate_CourseModules  
ON CourseModules(moduleDate)
```

13. Indeks na tabeli Courses dla startDate

Szybsze wyszukiwanie po dacie rozpoczęcia kursu

```
CREATE INDEX idx_startDate_Courses  
ON Courses(startDate)
```

14. Indeks na tabeli studies dla entryFeeProductID

Wyszukiwanie wpisowego dla danego produktu

```
CREATE INDEX idx_entryFeeProductID_studies  
ON studies(entryFeeProductID)
```

WIDOKI

vFinancialReports

Widok vFinancialReports służy do generowania raportów finansowych dotyczących produktów w systemie. Uwzględnia różne typy produktów, takie jak kursy, webinarze oraz studia, i oblicza całkowite przychody na podstawie zakończonych płatności (PaymentStatus=1)

```
SELECT p.productID,
       p.productName,
       CASE
           WHEN c.courseID IS NOT NULL THEN 'Kurs'
           WHEN w.webinarID IS NOT NULL THEN 'Webinar'
           WHEN s.studiesID IS NOT NULL THEN 'Studium'
           ELSE 'Inny Produkt'
       END AS ProductType,
       SUM(CASE
               WHEN op.PaymentStatus = 1 THEN p.price
               ELSE 0
           END) AS TotalRevenue
FROM products p
LEFT JOIN
    Courses c [1<->0..n] ON p.productID = c.productID
    LEFT JOIN
        Webinars w [1<->0..n] ON p.productID = w.productID
        LEFT JOIN
            studies s [1<->0..n] ON p.productID = s.entryFeeProductID -- Powiązanie wpisowego
            LEFT JOIN
                studySessions ss [1<->0..n] ON p.productID = ss.productID -- Powiązanie z opłatami za zajędy
                LEFT JOIN
                    orderDetails od [1<->0..n] ON p.productID = od.productID -- Powiązanie produktu z zamówieniem
                    LEFT JOIN
                        orders o [1..n<->1] ON od.orderID = o.orderID -- Powiązanie szczegółów zamówienia z zamówieniem
                        LEFT JOIN
                            OrderPayment op [1<->0..n] ON o.orderID = op.orderID -- Powiązanie zamówienia z płatnościami
GROUP BY p.productID,
         p.productName,
         CASE
             WHEN c.courseID IS NOT NULL THEN 'Kurs'
             WHEN w.webinarID IS NOT NULL THEN 'Webinar'
             WHEN s.studiesID IS NOT NULL THEN 'Studium'
             ELSE 'Inny Produkt'
         END
```

vDebtorsList

Widok vDebtorsList prezentuje listę użytkowników (dłużników), którzy mają niezapłacone zamówienia. Zawiera informacje o użytkownikach (ID, imię, nazwisko, email), szczegóły zamówienia (ID zamówienia, data zamówienia) oraz dane o produktach w zamówieniu (ID, nazwa produktu). Dodatkowo obliczana jest całkowita kwota zamówienia na podstawie cen produktów. Widok uwzględnia zamówienia, które nie zostały opłacone lub mają status płatności równy 0.

```
CREATE VIEW vDebtorsList AS
SELECT DISTINCT
    u.userID,
    u.firstName,
    u.lastName,
    u.email,
    o.orderID,
    p.productID,
    p.productName,
    o.orderDate,
    SUM(p.price) AS totalOrderAmount
FROM
    users u
INNER JOIN
    students s [1<->1] ON u.userID = s.studentID
INNER JOIN
    orders o [1<->1..n] ON s.studentID = o.studentID
INNER JOIN
    orderDetails od [1<->1..n] ON o.orderID = od.orderID
INNER JOIN
    products p [1..n<->1] ON od.productID = p.productID
LEFT JOIN
    OrderPayment op [1<->0..n] ON o.orderID = op.orderID
WHERE
    op.paymentID IS NULL
    OR op.PaymentStatus = 0
GROUP BY
    u.userID, u.firstName, u.lastName, u.email, o.orderID, p.productID, p.productName, o.orderDate;
```

vFutureEventsRegistrations

Widok vFutureEventRegistrations prezentuje informacje o przyszłych wydarzeniach w systemie (kursy, webinar, studium) z podziałem na:

Typ wydarzenia: Kurs, Webinar, Studium, lub Inny Produkt.

Tryb dostawy: Stacjonarnie (z przypisaną salą) lub Zdalnie.

Liczba uczestników: Zlicza unikalnych uczestników wydarzeń na podstawie odpowiednich tabel (CourseAccess, WebinarAccess, studiesParticipants, extraStudySessionsParticipants).

```

GROUP BY
    p.productID,
    p.productName,
    CASE
        WHEN c.courseID IS NOT NULL THEN 'Kurs'
        WHEN w.webinarID IS NOT NULL THEN 'Webinar'
        WHEN s.studiesID IS NOT NULL THEN 'Studium'
        ELSE 'Inny Produkt'
    END,
    CASE
        WHEN m.meetingID IS NOT NULL AND sm.room IS NOT NULL THEN 'Stacjonarnie'
        ELSE 'Zdalnie'
    END;
END;

SELECT
    p.productID,
    p.productName,
    CASE
        WHEN c.courseID IS NOT NULL THEN 'Kurs'
        WHEN w.webinarID IS NOT NULL THEN 'Webinar'
        WHEN s.studiesID IS NOT NULL THEN 'Studium'
        ELSE 'Inny Produkt'
    END AS ProductType,
    COUNT(DISTINCT CASE
        WHEN c.courseID IS NOT NULL THEN ca.studentID
        WHEN w.webinarID IS NOT NULL THEN wa.studentID
        WHEN s.studiesID IS NOT NULL THEN sp.studentID
        ELSE NULL
    END) AS TotalParticipants,
    CASE
        WHEN m.meetingID IS NOT NULL AND sm.room IS NOT NULL THEN 'Stacjonarnie'
        ELSE 'Zdalnie'
    END AS DeliveryMode
FROM
    products p
LEFT JOIN
    Courses c [1<->0..n] ON p.productID = c.productID
LEFT JOIN
    Webinars w [1<->0..n] ON p.productID = w.productID
LEFT JOIN
    studies s [1<->0..n] ON p.productID = s.entryFeeProductID
LEFT JOIN
    studySessions ss ON s.studiesID = ss.studiesID
LEFT JOIN
    extraStudySessionsParticipants essp ON ss.studiesID = essp.studiesID AND ss.meetingID = essp.meetingID
LEFT JOIN
    studiesParticipants sp [1<->0..n] ON s.studiesID = sp.studiesID
LEFT JOIN
    CourseAccess ca [1<->0..n] ON c.courseID = ca.courseID
LEFT JOIN
    WebinarAccess wa [1<->0..n] ON w.webinarID = wa.webinarID
LEFT JOIN
    meeting m [1..n<->1] ON ss.meetingID = m.meetingID
LEFT JOIN
    StationaryMeeting sm [1<->0..1] ON m.meetingID = sm.meetingID
WHERE
    m.startDate > GETDATE()

```

vCompletedEventsAttendance

Widok vCompletedEventAttendance przedstawia dane dotyczące zakończonych wydarzeń, takich jak kursy, webinary i studia, wraz z informacją o liczbie uczestników i obecnych. Uwzględnia on typ produktu, identyfikator spotkania, daty rozpoczęcia i zakończenia, a także zlicza unikalnych uczestników na podstawie tabel CourseAccess, WebinarAccess, studiesParticipants oraz extraStudySessionsParticipants. Dodatkowo obecność jest określana na podstawie tabeli meetingAttendance dla spotkań, jak również danych z powiązanych tabel uczestników. Widok zawiera jedynie wydarzenia, które zakończyły się przed bieżącą datą, i umożliwia analizę frekwencji oraz zaangażowania uczestników w różnych formach edukacyjnych

```

CREATE VIEW vCompletedEventAttendance AS
WITH ProductMeetings AS (
    SELECT
        p.productID,
        p.productName,
        CASE
            WHEN c.courseID IS NOT NULL THEN 'Kurs'
            WHEN w.webinarID IS NOT NULL THEN 'Webinar'
            WHEN s.studiesID IS NOT NULL THEN 'Studium'
            ELSE 'Inny Produkt'
        END AS ProductType,
        m.meetingID,
        m.startDate,
        m.endDate
    FROM
        products p
    LEFT JOIN
        Courses c [1<->0..n: ON p.productID = c.productID
    LEFT JOIN
        Webinars w [1<->0..n: ON p.productID = w.productID
    LEFT JOIN
        studies s [1<->0..n: ON p.productID = s.entryFeeProductID
    LEFT JOIN
        studySessions ss ON s.studiesID = ss.studiesID
    LEFT JOIN
        meeting m ON ss.meetingID = m.meetingID
            OR c.courseID = m.meetingID
            OR w.webinarID = m.meetingID
    WHERE
        m.endDate < GETDATE()
),
TotalParticipants AS (
    SELECT
        pm.productID,
        pm.meetingID,
        COUNT(DISTINCT
            CASE
                WHEN pm.ProductType = 'Kurs' THEN ca.studentID
                WHEN pm.ProductType = 'Webinar' THEN wa.studentID
                WHEN pm.ProductType = 'Studium' THEN sp.studentID
                ELSE NULL
            END
)
)
```

```

        END
    ) AS TotalParticipants
FROM
    ProductMeetings pm
LEFT JOIN
    Courses c ON pm.productID = c.productID
LEFT JOIN
    CourseAccess ca [ 1<->0..n: ON c.courseID = ca.courseID
LEFT JOIN
    Webinars w ON pm.productID = w.productID
LEFT JOIN
    WebinarAccess wa [ 1<->0..n: ON w.webinarID = wa.webinarID
LEFT JOIN
    studies s ON pm.productID = s.entryFeeProductID
LEFT JOIN
    studiesParticipants sp [ 1<->0..n: ON s.studiesID = sp.studiesID
GROUP BY
    pm.productID,
    pm.meetingID,
    pm.ProductType
),
TotalAttended AS (
SELECT
    pm.productID,
    pm.meetingID,
    COUNT(DISTINCT
        CASE
            WHEN pm.ProductType = 'Kurs' AND ma.attendance = 1 THEN ma.studentID
            WHEN pm.ProductType = 'Webinar' AND wa_att.studentID IS NOT NULL THEN wa_att.studentID
            WHEN pm.ProductType = 'Studium' AND sp.studentID IS NOT NULL THEN sp.studentID
            ELSE NULL
        END
    ) AS TotalAttended
FROM
    ProductMeetings pm
LEFT JOIN
    Courses c ON pm.productID = c.productID
LEFT JOIN
    CourseAccess ca [ 1<->0..n: ON c.courseID = ca.courseID
LEFT JOIN

```

```

        LEFT JOIN
            Webinars w ON pm.productID = w.productID
        LEFT JOIN
            WebinarAccess wa [1<->0..n] ON w.webinarID = wa.webinarID
        LEFT JOIN
            WebinarAttendance wa_att [1<->0..n] ON w.webinarID = wa_att.webinarID
        LEFT JOIN
            studies s ON pm.productID = s.entryFeeProductID
        LEFT JOIN
            studiesParticipants sp [1<->0..n] ON s.studiesID = sp.studiesID
        LEFT JOIN
            meetingAttendance ma ON pm.meetingID = ma.meetingID
    GROUP BY
        pm.productID,
        pm.meetingID,
        pm.ProductType
)
SELECT
    pm.productID,
    pm.productName,
    pm.ProductType,
    pm.meetingID,
    pm.startDate,
    pm.endDate,
    ISNULL(tp.TotalParticipants, 0) AS TotalParticipants,
    ISNULL(ta.TotalAttended, 0) AS TotalAttended
FROM
    ProductMeetings pm
LEFT JOIN
    TotalParticipants tp ON pm.productID = tp.productID AND pm.meetingID = tp.meetingID
LEFT JOIN
    TotalAttended ta ON pm.productID = ta.productID AND pm.meetingID = ta.meetingID
WHERE
    ISNULL(ta.TotalAttended, 0) <= ISNULL(tp.TotalParticipants, 0);

```

vAttendanceList

Widok vAttendanceList przedstawia listę obecności uczestników dla różnych wydarzeń, takich jak kursy, webinarze i studia. Zawiera informacje o produkcie, spotkaniu, uczestnikach (imię, nazwisko) oraz statusie ich obecności (Obecny lub Nieobecny).

```

1  SELECT
2      p.productID,
3      p.productName,
4      m.meetingID,
5      m.startDate,
6      m.endDate,
7      u.firstName,
8      u.lastName,
9      CASE
10         WHEN ma.attendance = 1 THEN 'Obecny'
11         ELSE 'Nieobecny'
12     END AS AttendanceStatus
13 FROM
14     products p
15 LEFT JOIN
16     Courses c [1<->0..n: ON p.productID = c.productID
17 LEFT JOIN
18     CourseAccess ca [1<->0..n: ON c.courseID = ca.courseID
19 LEFT JOIN
20     Webinars w [1<->0..n: ON p.productID = w.productID
21 LEFT JOIN
22     WebinarAccess wa [1<->0..n: ON w.webinarID = wa.webinarID
23 LEFT JOIN
24     studies s [1<->0..n: ON p.productID = s.entryFeeProductID
25 LEFT JOIN
26     studiesParticipants sp [1<->0..n: ON s.studiesID = sp.studiesID
27 LEFT JOIN
28     studySessions ss ON s.studiesID = ss.studiesID
29 LEFT JOIN
30     extraStudySessionsParticipants esp ON ss.studiesID = esp.studiesID AND ss.meetingID = esp.meetingID
31 INNER JOIN -- Zmienione z LEFT JOIN na INNER JOIN
32     meeting m ON ss.meetingID = m.meetingID OR c.courseID = m.meetingID OR w.webinarID = m.meetingID
33 LEFT JOIN
34     meetingAttendance ma [1<->0..n: ON m.meetingID = ma.meetingID
35 LEFT JOIN
36     WebinarAttendance wa_att ON w.webinarID = wa_att.webinarID AND wa_att.studentID = wa.studentID

```

```

LEFT JOIN
    students s_p ON (
        ma.studentID = s_p.studentID OR
        ca.studentID = s_p.studentID OR
        wa.studentID = s_p.studentID OR
        sp.studentID = s_p.studentID OR
        esp.studentID = s_p.studentID
    )
LEFT JOIN
    users u [1<->1: ON s_p.studentID = u.userID
WHERE
    m.endDate < GETDATE() OR w.webinarID IS NOT NULL

```

vBilocationReport

Widok BilocationReport_AllEvents generuje raport o uczestnikach zapisanych na co najmniej dwa przyszłe wydarzenia, których terminy czasowe kolidują. Wydarzenia mogą być spotkaniami (Meeting) lub webinarami (Webinar). Widok prezentuje identyfikator i dane osobowe uczestnika (imię, nazwisko), informacje o dwóch kolidujących wydarzeniach (typ, identyfikator, daty rozpoczęcia i

zakończenia). Dane są filtrowane tak, aby unikać powtarzania tych samych par wydarzeń oraz kolizji czasowych, a analizowane są wyłącznie przyszłe wydarzenia.

```
WITH AllEvents AS (
    SELECT
        ma.studentID,
        'Meeting' AS EventType,
        m.meetingID AS EventID,
        m.startDate,
        m.endDate
    FROM meeting m
    JOIN meetingAttendance ma
        1<->1..n: ON m.meetingID = ma.meetingID
    WHERE m.endDate > GETDATE()

    UNION ALL

    SELECT
        wa.studentID,
        'Webinar' AS EventType,
        w.webinarID AS EventID,
        w.startDate AS startDate,
        w.endDate AS endDate
    FROM Webinars w
    JOIN WebinarAttendance wa
        1<->1..n: ON w.webinarID = wa.webinarID
    WHERE w.endDate > GETDATE()
)
SELECT
    u.userID AS StudentID,
    u.firstName,
    u.lastName,
    e1.EventType AS EventType1,
    e1.EventID AS EventID1,
    e1.startDate AS StartDate1,
    e1.endDate AS EndDate1,
    e2.EventType AS EventType2,
    e2.EventID AS EventID2,
    e2.startDate AS StartDate2,
    e2.endDate AS EndDate2
FROM AllEvents e1
JOIN AllEvents e2
    ON e1.studentID = e2.studentID
```

vStudiesInternshipAttendance

Widok StudiesInternshipAttendance przedstawia informacje o uczestnictwie studentów w praktykach przypisanych do studiów. Zawiera szczegóły takie jak identyfikator praktyki, studiów, nazwę studiów, identyfikator i dane studenta (imię, nazwisko), liczbę wymaganych dni obecności (RequiredDays), liczbę faktycznie obecnych dni (AttendedDays) oraz status pełnej obecności (IsFullAttendance), który wskazuje, czy student był obecny przez wszystkie wymagane dni. Widok

wykorzystuje dane z tabel internship, internshipMeeting, studies oraz users, grupując dane na poziomie praktyki i studenta.

```
CREATE VIEW StudiesInternshipAttendance
AS
SELECT
    i.internshipID,
    i.studiesID,
    s.studiesName,
    im.studentID,
    u.firstName,
    u.lastName,
    (DATEDIFF(DAY, i.startDate, i.endDate) + 1) AS RequiredDays,
    COUNT(CASE WHEN im.attendance=1 THEN 1 END) AS AttendedDays,
    CASE
        WHEN COUNT(CASE WHEN im.attendance=1 THEN 1 END) =
            (DATEDIFF(DAY, i.startDate, i.endDate) + 1)
        THEN 1
        ELSE 0
    END AS IsFullAttendance
FROM internship i
JOIN studies s [1..n<->1] ON s.studiesID = i.studiesID
JOIN internshipMeeting im [1<->1..n] ON im.internshipID = i.internshipID
JOIN users u ON u.userID = im.studentID
GROUP BY
    i.internshipID,
    i.studiesID,
    s.studiesName,
    im.studentID,
    u.firstName,
    u.lastName,
    i.startDate,
    i.endDate;
GO
```

vCourseCompletionProgress

Widok przedstawia informacje o postępie studentów w ramach poszczególnych kursów. Zawiera szczegóły takie jak identyfikator i tytuł kursu, identyfikator studenta, imię i nazwisko, całkowitą liczbę modułów kursu (**TotalModules**),

liczbę zaliczonych modułów (`PassedModules`) oraz procent ukończenia kursu (`CompletionPercent`).

```
SELECT
    ca.courseID,
    co.title AS CourseTitle,
    ca.studentID,
    u.firstName,
    u.lastName,
    COUNT(*) AS TotalModules,
    COUNT(mp.moduleID) AS PassedModules,
    CASE
        WHEN COUNT(*)=0 THEN 0
        ELSE 100.0 * COUNT(mp.moduleID) / COUNT(*)
    END AS CompletionPercent
FROM CourseModules cm
JOIN Courses co 1..n<->1: ON co.courseID = cm.courseID
JOIN CourseAccess ca ON ca.courseID = cm.courseID
JOIN users u ON u.userID = ca.studentID
LEFT JOIN courseModulesPassed mp
    ON mp.moduleID = cm.moduleID
    AND mp.studentID = ca.studentID
GROUP BY
    ca.courseID,
    co.title,
    ca.studentID,
    u.firstName,
    u.lastName;
GO
```

vStudiesAttendancePercentag

Widok `vStudiesAttendancePercentage` prezentuje dane dotyczące frekwencji studentów na spotkaniach w ramach studiów. Zawiera identyfikator studiów, ich nazwę, identyfikator studenta oraz jego imię i nazwisko. Dodatkowo pokazuje liczbę wszystkich zaplanowanych spotkań (`TotalMeetings`), liczbę spotkań, w których student był obecny (`AttendedMeetings`), oraz procentową frekwencję (`AttendancePercent`). Pozwala on śledzić postęp zaliczenia danego studenta (wymagane 80% obecności).

```
CREATE VIEW vStudiesAttendancePercentage AS
SELECT
    sp.studiesID,
    st.studiesName,
    sp.studentID,
    u.firstName,
    u.lastName,
    COUNT(DISTINCT ss.meetingID) AS TotalMeetings,
    SUM(CASE WHEN ma.attendance = 1 THEN 1 ELSE 0 END) AS AttendedMeetings,
    CASE
        WHEN COUNT(DISTINCT ss.meetingID)=0 THEN 0
        ELSE 100.0 * SUM(CASE WHEN ma.attendance = 1 THEN 1 ELSE 0 END)
            / COUNT(DISTINCT ss.meetingID)
    END AS AttendancePercent
FROM studiesParticipants sp
JOIN studies st
    [1..n<->1] ON st.studiesID = sp.studiesID
JOIN users u
    ON u.userID = sp.studentID
LEFT JOIN studySessions ss
    ON ss.studiesID = sp.studiesID
LEFT JOIN meetingAttendance ma
    ON ma.meetingID = ss.meetingID
    AND ma.studentID = sp.studentID
GROUP BY
    sp.studiesID,
    st.studiesName,
    sp.studentID,
    u.firstName,
    u.lastName;
GO
```

10. CourseDiplomasView

Zadanie: po zakończeniu kursu należy wysłać dyplom uczestnikom pocztą. Widok ułatwia wylistowanie wszystkich osób, które **zaliczyły kurs** (czyli uczestniczyły w wymaganej liczbie modułów), oraz ich adres korespondencyjny.

```

CREATE VIEW CourseDiplomasView
AS
WITH Completion AS (
    SELECT
        ca.courseID,
        ca.studentID,
        COUNT(*) AS totalModules,
        SUM(CASE WHEN cmp.moduleID IS NOT NULL THEN 1 ELSE 0 END) AS passedModules
    FROM CourseAccess ca
    JOIN CourseModules cm ON cm.courseID = ca.courseID
    LEFT JOIN courseModulesPassed cmp
        ON cmp.moduleID = cm.moduleID
        AND cmp.studentID = ca.studentID
    GROUP BY
        ca.courseID,
        ca.studentID
)
SELECT
    co.courseID,
    co.title AS CourseTitle,
    comp.studentID,
    u.firstName,
    u.lastName,
    a.country,
    a.city,
    a.street,
    a.zipCode,
    CASE WHEN comp.totalModules = 0 THEN 0
        ELSE 100.0 * comp.passedModules / comp.totalModules
    END AS CompletionPercent
FROM Completion comp
JOIN Courses co
    ON co.courseID = comp.courseID
JOIN users u
    ON u.userID = comp.studentID
JOIN address a
    ON a.userID = comp.studentID
WHERE
    co.startDate < GETDATE()
    AND (CASE WHEN comp.totalModules = 0 THEN 0
        ELSE 100.0 * comp.passedModules / comp.totalModules
    END) >= 80.0;
GO

```

WebinarRecordingsView

Zadanie: baza nie przechowuje plików nagrani (są w systemie zewnętrznym), ale przechowuje linki. Widok pozwala łatwo przeglądać linki do nagrani i sprawdzać, do kiedy dany student ma dostęp (30 dni od webinaru).

```
CREATE VIEW WebinarRecordingsView
AS
SELECT
    w.webinarID,
    w.Title,
    w.recordingURL,
    wa.studentID,
    u.firstName,
    u.lastName,
    wa.accessStartDate,
    wa.accessEndDate
FROM Webinars w
JOIN WebinarAccess wa
    1<->1..n: ON w.webinarID = wa.webinarID
JOIN users u
    ON u.userID = wa.studentID;
GO
```

cartContentView

Zadanie: pokazuje zawartość koszyków zakupowych, czyli które produkty i ile sztuk. Ułatwia to zarządzanie (np. sprawdzenie, co studenci trzymają w koszyku i nie kupili).

```
CREATE VIEW CartContentView
AS
SELECT
    c.cartID,
    c.studentID,
    u.firstName,
    u.lastName,
    p.productID,
    p.productName,
    p.price
FROM cart c
JOIN cartDetails cd 1<->1..n: ON cd.cartID = c.cartID
JOIN products p 1..n<->1: ON p.productID = cd.productID
JOIN users u ON u.userID = c.studentID;
GO
```

revenueByMonth

Zadanie: raport finansowy przedstawiający przychód w ujęciu miesięcznym (suma wartości zamówień, które mają **PaymentStatus=1** i **paymentDate** wpada w dany miesiąc). Przydatny do monitorowania trendów sprzedaży.

```
CREATE VIEW RevenueByMonthView
AS
SELECT
    YEAR(op.paymentDate) AS PaymentYear,
    MONTH(op.paymentDate) AS PaymentMonth,
    SUM(p.price) AS MonthlyRevenue
FROM OrderPayment op
JOIN orders o
    1..n<->1: ON o.orderID = op.orderID
JOIN orderDetails od
    1<->1..n: ON od.orderID = o.orderID
JOIN products p
    1..n<->1: ON p.productID = od.productID
WHERE op.PaymentStatus = 1
GROUP BY
    YEAR(op.paymentDate),
    MONTH(op.paymentDate);
GO
```

UPRAWNIENIA I ROLE W SYSTEMIE

Użytkownicy systemu:

- 1.Uczestnik
- 2.Wykładowca
- 3.Administrator

Uprawnienia użytkowników:

1.Uczestnik

```
GRANT SELECT ON studies TO Participant
GRANT SELECT ON studySessions TO Participant
GRANT SELECT ON internship TO Participant
GRANT SELECT ON studyModules TO Participant
GRANT SELECT ON Webinars TO Participant
GRANT SELECT ON WebinarAttendance TO Participant
GRANT SELECT ON Courses TO Participant
GRANT SELECT ON courseModulesPassed TO Participant
GRANT SELECT ON CourseModules TO Participant
GRANT SELECT ON meeting TO Participant
GRANT SELECT ON meetingAttendance TO Participant
GRANT SELECT ON Substitution TO Participant
GRANT SELECT ON Rooms TO Participant
GRANT SELECT ON StationaryMeeting TO Participant
GRANT SELECT ON orders TO Participant
GRANT SELECT ON orderDetails TO Participant
GRANT SELECT ON products TO Participant
GRANT SELECT ON cart TO Participant
GRANT SELECT ON cartDetails TO Participant
GRANT SELECT ON currencies TO Participant

GRANT EXECUTE ON availableSeatsForCourses TO Participant
GRANT EXECUTE ON availableSeatsForStudies TO Participant
GRANT EXECUTE ON availableSeatsForStudiesSession TO Participant
GRANT EXECUTE ON getShoppingCartValue TO Participant
GRANT EXECUTE ON isTranslatorAssignedToMeeting TO Participant
GRANT EXECUTE ON orderCreate TO Participant
GRANT EXECUTE ON productCartAdd TO Participant
GRANT EXECUTE ON remainingWebinarAccessDays TO Participant
GRANT EXECUTE ON userCourseAttendancePercentage TO Participant
GRANT EXECUTE ON studentModifyAddress TO Participant
```

Instrukcje SQL typu GRANT, które przypisują uprawnienia użytkownikowi o nazwie Participant. Użytkownik ten otrzymuje prawa do przeglądania (SELECT) danych z różnych tabel, takich jak studies, studySessions, czy Webinars. Dodatkowo, nadano mu uprawnienia do wykonywania (EXECUTE) określonych procedur, takich jak availableSeatsForCourses, getShoppingCartValue czy orderCreate. Całość wskazuje na konfigurację dostępu do bazy danych, zapewniając użytkownikowi Participant możliwość korzystania z określonych funkcjonalności systemu.

2.Administrator

```
GRANT ALL PRIVILEGES TO Admin
```

polecenie SQL GRANT ALL PRIVILEGES TO Admin. Oznacza to, że użytkownik Admin otrzymuje pełne uprawnienia do bazy danych, w tym możliwość wykonywania wszystkich operacji na każdej tabeli, widoku czy procedurze. Admin ma dostęp zarówno do zarządzania danymi (wstawianie, modyfikowanie, usuwanie, odczyt), jak i administracji bazą (np. nadawanie uprawnień innym użytkownikom).

3.Wykładowca

```

GRANT SELECT ON students          TO Lecturer;
GRANT SELECT ON users            TO Lecturer;
GRANT SELECT ON translators      TO Lecturer;
GRANT SELECT ON languages         TO Lecturer;
GRANT SELECT ON coordinators     TO Lecturer;
GRANT SELECT ON courses          TO Lecturer;
GRANT SELECT ON courseModules    TO Lecturer;
GRANT SELECT, INSERT, UPDATE ,DELETE ON courseModulesPassed           TO Lecturer;
GRANT SELECT ON meeting          TO Lecturer;
GRANT SELECT, INSERT, UPDATE ,DELETE ON meetingAttendance TO Lecturer;
GRANT SELECT, INSERT, UPDATE ,DELETE ON StationaryMeeting           TO Lecturer;
GRANT SELECT ON Rooms            TO Lecturer;
GRANT SELECT ON Substitution     TO Lecturer;
GRANT SELECT ON studyModules     TO Lecturer;
GRANT SELECT ON studies          TO Lecturer;
GRANT SELECT ON studySessions    TO Lecturer;
GRANT SELECT, INSERT, UPDATE ,DELETE ON extraStudySessionsParticipants TO Lecturer;
GRANT SELECT ON studyModuleParticipants   TO Lecturer;
GRANT SELECT ON internship        TO Lecturer;
GRANT SELECT ON internshipMeeting   TO Lecturer;
GRANT SELECT ON studiesParticipants TO Lecturer
GRANT SELECT ON Webinars          TO Lecturer
GRANT SELECT, INSERT, UPDATE ,DELETE ON WebinarAttendance TO Lecturer
GRANT EXECUTE ON meetingPresence TO Lecturer
GRANT EXECUTE ON AddStudentToCourseModulesPassed TO Lecturer
GRANT EXECUTE ON userCourseAttendancePercentage TO Lecturer
GRANT EXECUTE ON studyAttendancePercentage TO Lecturer

```

W drugiej części przedstawiono szczegółowe przypisanie uprawnień dla użytkownika Lecturer. Uprawnienia te obejmują:

- Dostęp tylko do odczytu (SELECT) do tabel takich jak students, translators, courses, czy studyModules.
- Pełne operacje na danych (SELECT, INSERT, UPDATE, DELETE) na tabelach związanych z aktywnościami wykładowców, np. courseModulesPassed, meetingAttendance, StationaryMeeting, czy extraStudySessionsParticipants.
- Możliwość wykonania procedur (EXECUTE), takich jak meetingPresence, AddStudentToCourseModulesPassed, userCourseAttendancePercentage, czy studyAttendancePercentage.

GENEROWANIE DANYCH

Skrypt Python łączy się z bazą danych SQL Server za pomocą [pyodbc](#) i wykorzystuje bibliotekę [Faker](#) do generowania realistycznych danych syntetycznych. Najpierw tworzy dane w tabelach pomocniczych takich jak języki, waluty i sale. Następnie generuje użytkowników, przypisuje im role (tutorzy, tłumacze, koordynatorzy, studenci) oraz adresy i zgody RODO. Kolejne kroki obejmują tworzenie produktów (kursów, webinarów, studiów), modułów, spotkań oraz powiązanych rekordów uczestnictwa i frekwencji. Skrypt zapewnia integralność danych poprzez odpowiednią kolejność wstawiania rekordów i użycie unikalnych identyfikatorów, a na końcu zatwierdza wszystkie zmiany w bazie danych.

```
1  < import pyodbc
2  from faker import Faker
3  import random
4  import datetime
5
6  # KONFIGURACJA POŁĄCZENIA
7  #####
8  #####
9  connection_string = (
10 <     "DRIVER={ODBC Driver 17 for SQL Server};"
11     "SERVER=dbmanage.lab.ii.agh.edu.pl;"
12     "DATABASE=u_sbarczyk;"
13     "UID=u_sbarczyk;"
14     "PWD=TMeFpcGmGTr0;"
15 )
16
17 # PARAMETRY OGÓLNE I LICZBOWE
18 #####
19 #####
20
21 faker = Faker(["pl_PL"])
22 random.seed(42)
23
24 # Łącznie 200 użytkowników:
25 NUM_USERS = 200
26
27 # Rozkład ról:
28 NUM_TUTORS = 25
29 NUM_TRANSLATORS = 10
30 NUM_COORDINATORS = 15
31 NUM_STUDENTS = 150 # Suma = 200
32
33 NUM_LANGUAGES = 10
34 NUM_CURRENCIES = 5
35 NUM_ROOMS = 20
36 NUM_RODO = 150
37 NUM_ADDRESS = 200 # Każdy user ma 1 adres
38
39 # Produkty:
40 NUM_COURSES = 40
41 NUM_WEBINARS = 50
42 NUM_STUDIES = 30
```

```
NUM_STUDY_SESSIONS = 100

# Moduły:
NUM_COURSE_MODULES = 100 # W CourseModules moduleID nie jest IDENTITY
NUM_STUDY_MODULES = 80    # W studyModules moduleID jest IDENTITY

# Spotkania:
NUM_MEETINGS = 150
NUM_STATIONARY_MEETINGS = 30
NUM_ONLINE_SYNC_MEETINGS = 40
NUM_ONLINE_ASYNC_MEETINGS = 30

# Ile powiązań w moduleMeetings (jeśli PK jest (meetingID), nie wolno powtarzać meetingID):
NUM_MODULE_MEETINGS = 100 # Dostosowane do unikalności meetingID

# **Trzykrotnie zwiększone** wielokrotne wpisy:
NUM_MEETING_ATTENDANCE = 1500
NUM_WEBINAR_ATTENDANCE = 720
NUM_COURSE_MODULES_PASSED = 1200

NUM_WEBINAR_ACCESS = 80
NUM_COURSE_ACCESS = 100
NUM_STUDIES_PARTICIPANTS = 100
NUM_STUDY_MODULE_PARTICIPANTS = 100
NUM_INTERNSHIP = 20
NUM_INTERNSHIP_MEETING = 80
NUM_STUDENT_ACHIEVEMENTS = 60

# Koszyki i zamówienia:
NUM_CARTS = 100
NUM_CART_DETAILS = 150
NUM_ORDERS = 120
NUM_ORDER_DETAILS = 200
NUM_ORDER_PAYMENTS = 150

# Historia cen:
NUM_PRICE_HISTORY_PER_PRODUCT = 2
NUM_SUBSTITUTION = 25

#####
# FUNKCJE POMOCNICZE                                #
#####
```

```
def random_date(start_year=2020, end_year=2025):
    start_date = datetime.date(start_year, month: 1, day: 1)
    end_date = datetime.date(end_year, month: 12, day: 31)
    delta = (end_date - start_date).days
    rand_days = random.randint( a: 0, delta)
    return start_date + datetime.timedelta(days=rand_days)

def random_datetime(start_year=2022, end_year=2025):
    d = random_date(start_year, end_year)
    hour = random.randint( a: 8, b: 20)
    minute = random.randint( a: 0, b: 59)
    return datetime.datetime(d.year, d.month, d.day, hour, minute)

def insert_many(cursor, table_name, columns, values):
    placeholders = "(" + ",".join(["?"] * len(columns)) + ")"
    col_string = "(" + ",".join(columns) + ")"
    sql = f"INSERT INTO {table_name} {col_string} VALUES {placeholders}"
    cursor.executemany(sql, values)

#####
# GŁÓWNA FUNKCJA
#####
def main():
    conn = pyodbc.connect(connection_string)
    cursor = conn.cursor()

    # =====
    # 1. Tabele "pomocnicze": languages, currencies, rooms
    # =====
    language_pool = [
        "Polski", "Angielski", "Niemiecki", "Francuski", "Hiszpański",
        "Włoski", "Rosyjski", "Ukraiński", "Chiński", "Japoński",
        "Portugalski", "Czeski", " Słowacki", "Litewski"
    ]
    random.shuffle(language_pool)
    lang_inserts = []
    for i in range(NUM_LANGUAGES):
        lang_id = i + 1
        name = language_pool[i % len(language_pool)]
        lang_inserts.append((lang_id, name))
    insert_many(cursor, table_name: "languages", columns: ("id", "name"), lang_inserts)
```

```

# Waluty (realistyczne kursy względem PLN)
real_currencies = [
    ("PLN", 1.0),
    ("EUR", 4.5),
    ("USD", 4.0),
    ("GBP", 5.2),
    ("CHF", 4.6)
]
insert_many(cursor, table_name: "currencies", columns: ("currencyName", "currencyRate"), real_currencies)

# ROOMS
room_inserts = []
for _ in range(NUM_ROOMS):
    rname = f"Sala {random.randint(a: 1, b: 999)}"
    limit = random.randint(a: 15, b: 120)
    room_inserts.append((rname, limit))
insert_many(cursor, table_name: "Rooms", columns: ("name", "placeLimit"), room_inserts)

cursor.execute("SELECT id FROM Rooms ORDER BY id")
all_room_ids = [row.id for row in cursor.fetchall()]

# =====
# 2. Users (200) i role
# =====
users_inserts = []
# Generujemy e-mail pasujący do imienia i nazwiska
for i in range(NUM_USERS):
    fname = faker.first_name()
    lname = faker.last_name()
    # Stworzymy email = imię.nazwisko{i}@example.com
    email_local = f"{fname.lower()}.{lname.lower()}@{i}"
    mail = email_local + "@example.com"
    pwd = faker.password(length=8)
    users_inserts.append((fname, lname, mail, pwd))

insert_many(cursor, table_name: "users", columns: ("firstName", "lastName", "email", "password"), users_inserts)

cursor.execute("SELECT userID FROM users ORDER BY userID")
all_user_ids = [row.userID for row in cursor.fetchall()]

# Rozdzielimy role w sposób deterministyczny:
tutor_ids = all_user_ids[0:NUM_TUTORS]

```

```

# Rozdzielamy role w sposób deterministyczny:
tutor_ids = all_user_ids[0:NUM_TUTORS]
translator_ids = all_user_ids[NUM_TUTORS : NUM_TUTORS + NUM_TRANSLATORS]
coordinator_ids = all_user_ids[NUM_TUTORS + NUM_TRANSLATORS : NUM_TUTORS + NUM_TRANSLATORS + NUM_COORDINATORS]
student_ids = all_user_ids[NUM_TUTORS + NUM_TRANSLATORS + NUM_COORDINATORS : ]

# TUTORS
insert_many(cursor, table_name: "tutors", columns: ("tutorID",), [(t,) for t in tutor_ids])

# TRANSLATORS
translator_inserts = []
for t in translator_ids:
    lid = random.randint(0, NUM_LANGUAGES)
    translator_inserts.append((t, lid))
insert_many(cursor, table_name: "translators", columns: ("translatorID", "languageID"), translator_inserts)

# COORDINATORS
insert_many(cursor, table_name: "coordinators", columns: ("coordinatorID",), [(c,) for c in coordinator_ids])

# STUDENTS
insert_many(cursor, table_name: "students", columns: ("studentID",), [(s,) for s in student_ids])

# =====
# 3. address (każdy user ma 1 adres, w Polsce) + RODO
# =====
address_inserts = []
for uid in all_user_ids:
    country = "Polska"
    city = faker.city()
    street = f"{faker.street_name()} {random.randint(0, b: 100)}"
    zipcode = f"{random.randint(0, b: 99):02d}-{random.randint(0, b: 999):03d}"
    address_inserts.append((uid, country, city, street, zipcode))
insert_many(cursor, table_name: "address", columns: ("userID", "country", "city", "street", "zipCode"), address_inserts)

rodo_inserts = []
for i in range(NUM_RODO):
    consent_id = i + 1
    uid = random.choice(all_user_ids)
    status = random.randint(0, 1)
    ts = random_date(start_year: 2020, end_year: 2025)
    rodo_inserts.append((consent_id, uid, status, ts))

```

```
    insert_many(cursor, table_name: "RODO", columns: ("consent_id", "userID", "status", "timestamp"), rodo_inserts)

    # =====#
    # FUNKCJA: Wstaw produkt i zwróć productID
    # =====#
    def create_product_return_id(name, price, update_date):
        sql = """
            INSERT INTO products (price, productName, priceUpdateDate)
            OUTPUT INSERTED.productID
            VALUES (?, ?, ?)
        """
        cursor.execute(sql, *params: (price, name, update_date))
        row = cursor.fetchone()
        if row is None or row[0] is None:
            raise Exception("INSERT do products się nie powiodł!")
        return int(row[0])

    # =====#
    # 4. Generujemy 3 rodzaje produktów: Kursy, Webinary, Studia
    # =====#
    course_titles_pool = [
        "Kurs Python - od podstaw", "Kurs Java dla Początkujących",
        "Kurs Data Science", "Kurs SQL i Bazy Danych",
        "Kurs Excel Zaawansowany", "Kurs Analiza Danych w Pythonie",
        "Kurs Frontend (HTML/CSS/Javascript)", "Kurs C++ Embedded",
        "Kurs WordPress", "Kurs Docker i Kubernetes",
        "Kurs Machine Learning", "Kurs Statystyka dla Programistów"
    ]
    webinar_titles_pool = [
        "Webinar Marketing Online", "Webinar Data Visualization",
        "Webinar Scrum w Praktyce", "Webinar RPA w Firmie",
        "Webinar Big Data w Chmurze", "Webinar Podstawy ML",
        "Webinar Bezpieczeństwo w Sieci", "Webinar JavaScript nowości ES6+",
        "Webinar DevOps - wprowadzenie"
    ]
    study_names_pool = [
        "Informatyka Stosowana", "Zarządzanie", "Mechanika i Budowa Maszyn",
        "Elektronika i Telekomunikacja", "Automatyka i Robotyka", "Matematyka",
        "Filologia Angielska", "Ekonomia", "Energetyka", "Sztuczna Inteligencja",
        "Fizyka Techniczna", "Chemia Medyczna"
    ]
```

```
course_product_ids = []
webinar_product_ids = []
studies_product_ids = []

# Kursy:
chosen_course_titles = []
for _ in range(NUM_COURSES):
    base_title = random.choice(course_titles_pool)
    ctitle = f'{base_title} - edycja {random.randint(a: 1, b: 99)}'
    chosen_course_titles.append(ctitle)

for title in chosen_course_titles:
    price = round(random.uniform(a: 100, b: 3000), 2)
    pdate = random_datetime(start_year: 2022, end_year: 2024)
    pid = create_product_return_id(title, price, pdate)
    course_product_ids.append(pid)

# Webinary:
chosen_webinar_titles = []
for _ in range(NUM_WEBINARS):
    wtitle = random.choice(webinar_titles_pool)
    wtitle += f' (edycja {random.randint(a: 1, b: 50)})'
    chosen_webinar_titles.append(wtitle)

for wtitle in chosen_webinar_titles:
    price = round(random.uniform(a: 0, b: 800), 2)
    pdate = random_datetime(start_year: 2022, end_year: 2024)
    pid = create_product_return_id(wtitle, price, pdate)
    webinar_product_ids.append(pid)

# Studia (wpisowe):
chosen_study_titles = []
for _ in range(NUM_STUDIES):
    sname = random.choice(study_names_pool)
    sname += f' (rok {random.randint(a: 1, b: 5)})'
    chosen_study_titles.append(sname)

for stitle in chosen_study_titles:
    price = round(random.uniform(a: 500, b: 5000), 2)
    pdate = random_datetime(start_year: 2022, end_year: 2024)
    pid = create_product_return_id("Wpisowe: " + stitle, price, pdate)
    studies_product_ids.append(pid)
```

```

# =====
# 5. Tabela Courses
# =====
courses_inserts = []
cursor.execute("SELECT coordinatorID FROM coordinators ORDER BY coordinatorID")
all_coordinator_ids = [row.coordinatorID for row in cursor.fetchall()] # ewentualnie to samo co coordinator_ids

for i in range(NUM_COURSES):
    pid = course_product_ids[i]
    coord_id = random.choice(all_coordinator_ids)
    title = chosen_course_titles[i]
    desc = f"Kurs «{title}». Nauka od podstaw do zagadnień zaawansowanych."
    stdate = random_date( start_year: 2023, end_year: 2025)
    courses_inserts.append((pid, coord_id, title, desc, stdate))

insert_many(cursor, table_name: "Courses",
           columns: ("productID","coordinatorID","title","description","startDate"),
           courses_inserts)

# =====
# 6. Tabela Webinars
# =====
webinar_inserts = []
for i in range(NUM_WEBINARS):
    pid = webinar_product_ids[i]
    t_id = random.choice(tutor_ids)
    tr_id = random.choice(translator_ids) if random.random() < 0.3 else None
    wtitle = chosen_webinar_titles[i]
    sdate = random_datetime( start_year: 2023, end_year: 2025)
    edate = sdate + datetime.timedelta(hours=2)
    desc = f"Opis webinaru: {wtitle}"
    rec_url = "https://webinar-recordings.fake/" + str(random.randint( a: 1000, b: 9999))
    lang = random.randint( a: 1, NUM_LANGUAGES)
    webinar_inserts.append((pid, t_id, tr_id, wtitle, sdate, edate, desc, rec_url, lang))

insert_many(cursor, table_name: "Webinars",
           columns: ("productID","tutorID","translatorID","Title","StartDate","endDate",
                     "Description","recordingURL","languageID"),
           webinar_inserts
)

```

```

# =====
# 7. Tabela studies
# =====
studies_inserts = []
for i in range(NUM_STUDIES):
    plimit = random.randint( a: 10, b: 80)
    sname = chosen_study_titles[i]
    entry_fee_pid = studies_product_ids[i]
    studies_inserts.append((plimit, sname, entry_fee_pid))
insert_many(cursor, table_name: "studies",
            columns: ("placeLimit", "studiesName", "entryFeeProductID"),
            studies_inserts)

# =====
# 8. CourseModules (moduleID bez IDENTITY)
# =====
cursor.execute("SELECT courseID FROM Courses ORDER BY courseID")
course_ids_list = [row.courseID for row in cursor.fetchall()]

module_titles_for_courses = [
    "Podstawy", "Wprowadzenie do składni", "Struktury danych",
    "Zaawansowane techniki", "Testy jednostkowe", "Optymalizacja kodu",
    "Projekt końcowy", "Przykłady z biblioteki standardowej"
]
course_modules_inserts = []
module_id_counter = 1
for _ in range(NUM_COURSE_MODULES):
    c_id = random.choice(course_ids_list)
    t_id = random.choice(tutor_ids)
    mtitle = random.choice(module_titles_for_courses)
    mdate = random_datetime( start_year: 2023, end_year: 2025)
    lang = random.randint( a: 1, NUM_LANGUAGES)
    course_modules_inserts.append((module_id_counter, c_id, t_id, mtitle, mdate, lang))
    module_id_counter += 1

insert_many(cursor, table_name: "CourseModules",
            columns: ("moduleID", "courseID", "tutorID", "moduleTitle", "moduleDate", "languageID"),
            course_modules_inserts)

# =====
# 9. studyModules (moduleID = IDENTITY)
# =====

```

```

    cursor.execute("SELECT studiesID FROM studies ORDER BY studiesID")
    studies_ids_list = [row.studiesID for row in cursor.fetchall()]

    module_titles_for_studies = [
        "Wprowadzenie do kierunku", "Zagadnienia zaawansowane", "Projekt semestralny",
        "Laboratorium praktyczne", "Seminarium badawcze", "Warsztaty specjalistyczne",
        "Egzamin próbny", "Analiza przypadków"
    ]
    study_modules_inserts = []
    for _ in range(NUM_STUDY_MODULES):
        sid = random.choice(studies_ids_list)
        coord_id = random.choice(coordinator_ids)
        t = random.choice(module_titles_for_studies)
        d = f"Opis modułu: {t}"
        study_modules_inserts.append((sid, t, d, coord_id))
    insert_many(cursor, table_name: "studyModules",
                columns: ("studiesID", "moduleTitle", "description", "coordinatorID"),
                study_modules_inserts)

# =====
# 10. meeting
# =====
meeting_inserts = []
for _ in range(NUM_MEETINGS):
    tut_id = random.choice(tutor_ids)
    sd = random_datetime( start_year: 2023, end_year: 2025)
    ed = sd + datetime.timedelta(hours=random.randint( a: 1, b: 4))
    lang = random.randint( a: 1, NUM_LANGUAGES)
    tr_id = random.choice(translator_ids) if random.random() < 0.2 else None
    meeting_inserts.append((tut_id, sd, ed, lang, tr_id))
insert_many(cursor, table_name: "meeting",
            columns: ("tutorID", "startDate", "endDate", "languageID", "meetingTranslator"),
            meeting_inserts)

cursor.execute("SELECT meetingID FROM meeting ORDER BY meetingID")
all_meeting_ids = [row.meetingID for row in cursor.fetchall()]

# =====
# 11. studySessions -> osobny produkt "Sesja ..."
# =====
cursor.execute("SELECT studiesID, moduleID FROM studyModules ORDER BY studiesID, moduleID")
all_study_modules = [(row.studiesID, row.moduleID) for row in cursor.fetchall()]

```

```

study_session_inserts = []
study_session_product_map = []
num_sessions_to_generate = min(NUM_STUDY_SESSONS, len(all_meeting_ids))

def create_study_session_product(studies_id, module_id):
    prod_name = f"Sesja (studia {studies_id}, moduł {module_id})"
    price = round(random.uniform(a: 50, b: 600), 2)
    pdate = random_datetime(start_year: 2022, end_year: 2024)
    return create_product_return_id(prod_name, price, pdate)

for _ in range(num_sessions_to_generate):
    sid, modid = random.choice(all_study_modules)
    mid = random.choice(all_meeting_ids)
    p_id = create_study_session_product(sid, modid)
    study_session_product_map.append((sid, modid, mid, p_id))

for (sid, modid, mid, p_id) in study_session_product_map:
    study_session_inserts.append((mid, p_id, sid, modid))

insert_many(cursor, table_name: "studySessions",
           columns: ("meetingID", "productID", "studiesID", "moduleID"),
           study_session_inserts)

# Zbieramy wszystkie productID
all_product_ids = []
all_product_ids.extend(course_product_ids)
all_product_ids.extend(webinar_product_ids)
all_product_ids.extend(studies_product_ids)
for (_, _, _, p) in study_session_product_map:
    all_product_ids.append(p)

# =====
# 12. Stationary, OnlineSync, onlineAsync
# =====
chosen_stationary = random.sample(all_meeting_ids, min(NUM_STATIONARY_MEETINGS, len(all_meeting_ids)))
st_inserts = []
for mid in chosen_stationary:
    room_id = random.choice(all_room_ids)
    st_inserts.append((mid, room_id))
insert_many(cursor, table_name: "StationaryMeeting", columns: ("meetingID", "room"), st_inserts)

```

```

remain_after_stat = list(set(all_meeting_ids) - set(chosen_stationary))
chosen_sync = random.sample(remain_after_stat, min(NUM_ONLINE_SYNC_MEETINGS, len(remain_after_stat)))
sync_inserts = []
for mid in chosen_sync:
    link = f"https://live-sync.fake/{random.randint(a: 1000, b: 9999)}"
    sync_inserts.append((mid, link))
insert_many(cursor, table_name: "OnlineSyncMeeting", columns: ("meetingID", "link"), sync_inserts)

remain_after_sync = list(set(remain_after_stat) - set(chosen_sync))
chosen_async = random.sample(remain_after_sync, min(NUM_ONLINE_ASYNC_MEETINGS, len(remain_after_sync)))
async_inserts = []
for mid in chosen_async:
    url = f"https://async.fake/{random.randint(a: 1000, b: 9999)}"
    async_inserts.append((mid, url))
insert_many(cursor, table_name: "onlineAsyncMeeting", columns: ("meetingID", "recordingURL"), async_inserts)

# =====
# 13. moduleMeetings (z kluczem PK(meetingID) => 1 spotkanie = 1 moduł)
#   Ograniczamy się do `NUM_MODULE_MEETINGS` unikatowych meetingID.
# =====
# Ponieważ PK jest (meetingID), nie możemy dać 2 wierszy z tym samym meetingID.
# Wybieramy losowo "NUM_MODULE_MEETINGS" meetingów i przypisujemy je do modułów.
cursor.execute("SELECT moduleID FROM CourseModules ORDER BY moduleID")
all_course_module_ids = [row.moduleID for row in cursor.fetchall()]

random.shuffle(all_meeting_ids)
limit_mm = min(NUM_MODULE_MEETINGS, len(all_meeting_ids))
module_meetings_inserts = []
for i in range(limit_mm):
    # 1 do 1
    m_id = all_meeting_ids[i]
    mod_id = random.choice(all_course_module_ids)
    # kolumny w moduleMeetings: (moduleID, meetingID)
    module_meetings_inserts.append((mod_id, m_id))

insert_many(cursor, table_name: "moduleMeetings", columns: ("moduleID", "meetingID"), module_meetings_inserts)

# =====
# 14. meetingAttendance (1500 rekordów), Substitution
# =====
ma_pairs = set()
while len(ma_pairs) < NUM_MEETING_ATTENDANCE:

```

```

# =====
# 14. meetingAttendance (1500 rekordów), Substitution
# =====
ma_pairs = set()
while len(ma_pairs) < NUM_MEETING_ATTENDANCE:
    mid = random.choice(all_meeting_ids)
    sid = random.choice(student_ids)
    ma_pairs.add((sid, mid))

ma_inserts = []
for (sid, mid) in ma_pairs:
    att = random.randint(a: 0, b: 1)
    ma_inserts.append((sid, mid, att))
insert_many(cursor, table_name: "meetingAttendance",
           columns: ("studentID", "meetingID", "attendance"),
           ma_inserts)

subs_inserts = []
for _ in range(NUM_SUBSTITUTION):
    mid = random.choice(all_meeting_ids)
    subt = random.choice(tutor_ids)
    reason = "Zastępstwo spowodowane nagłą nieobecnością"
    subs_inserts.append((mid, subt, reason))
insert_many(cursor, table_name: "Substitution", columns: ("meetingID", "substituteTutorID", "reason"), subs_inserts)

# =====
# 15. ProductPriceHistory
# =====
pph_inserts = []
for pid in all_product_ids:
    for _ in range(NUM_PRICE_HISTORY_PER_PRODUCT):
        sd = random_datetime(start_year: 2021, end_year: 2022)
        ed = sd + datetime.timedelta(days=random.randint(a: 30, b: 300))
        old_price = round(random.uniform(a: 50, b: 5000), 2)
        pph_inserts.append((pid, old_price, sd, ed))
insert_many(cursor, table_name: "ProductPriceHistory",
           columns: ("productID", "price", "startDate", "endDate"),
           pph_inserts)

# =====
# 16. Koszyki i Zamówienia

```

```

# =====
cart_inserts = []
for _ in range(NUM_CARTS):
    stid = random.choice(student_ids)
    cart_inserts.append((stid,))
insert_many(cursor, table_name: "cart", columns: ("studentID",), cart_inserts)

cursor.execute("SELECT cartID FROM cart ORDER BY cartID")
all_cart_ids = [row.cartID for row in cursor.fetchall()]

cart_details_inserts = set()
while len(cart_details_inserts) < NUM_CART_DETAILS:
    c_id = random.choice(all_cart_ids)
    p_id = random.choice(all_product_ids)
    cart_details_inserts.add((c_id, p_id))
insert_many(cursor, table_name: "cartDetails", columns: ("cartID","productID"), list(cart_details_inserts))

orders_inserts = []
for _ in range(NUM_ORDERS):
    stid = random.choice(student_ids)
    o_date = random_datetime( start_year: 2022, end_year: 2025)
    orders_inserts.append((stid, o_date))
insert_many(cursor, table_name: "orders", columns: ("studentID","orderDate"), orders_inserts)

cursor.execute("SELECT orderId FROM orders ORDER BY orderId")
all_order_ids = [row.orderID for row in cursor.fetchall()]

order_details_inserts = set()
while len(order_details_inserts) < NUM_ORDER_DETAILS:
    o_id = random.choice(all_order_ids)
    p_id = random.choice(all_product_ids)
    order_details_inserts.add((o_id, p_id))
insert_many(cursor, table_name: "orderDetails", columns: ("orderId","productID"), list(order_details_inserts))

payment_inserts = []
for _ in range(NUM_ORDER_PAYMENTS):
    exc = random.randint( a: 0, b: 1)
    status = random.randint( a: 0, b: 1)
    adv = random.randint( a: 0, b: 1)
    oid = random.choice(all_order_ids)
    p_date = random_datetime( start_year: 2022, end_year: 2025)
    link = f"https://platnosci.fake/{random.randint( a: 1000, b: 9999)}"

```

```

    payment_inserts.append((exc, status, oid, adv, p_date, link))
    insert_many(cursor, table_name: "OrderPayment",
        columns: ("exception", "PaymentStatus", "orderID", "advance", "paymentDate", "link"),
        payment_inserts)

# =====
# 17. CourseAccess, courseModulesPassed (zwiększone)
# =====
cursor.execute("SELECT courseID FROM Courses ORDER BY courseID")
all_course_ids = [row.courseID for row in cursor.fetchall()]

ca_pairs = set()
while len(ca_pairs) < NUM_COURSE_ACCESS:
    c_id = random.choice(all_course_ids)
    s_id = random.choice(student_ids)
    ca_pairs.add((c_id, s_id))

ca_inserts = []
for (c_id, s_id) in ca_pairs:
    start_d = random_datetime(start_year: 2023, end_year: 2025)
    end_d = start_d + datetime.timedelta(days=30)
    ca_inserts.append((c_id, s_id, start_d, end_d))
insert_many(cursor, table_name: "CourseAccess",
    columns: ("courseID", "studentID", "accessStartDate", "accessEndDate"),
    ca_inserts)

cursor.execute("SELECT moduleID, courseID FROM CourseModules")
all_course_modules = [(row.moduleID, row.courseID) for row in cursor.fetchall()]

cmp_inserts = set()
while len(cmp_inserts) < NUM_COURSE_MODULES_PASSED:
    mod_id, c_id = random.choice(all_course_modules)
    s_id = random.choice(student_ids)
    cmp_inserts.add((mod_id, s_id))
insert_many(cursor, table_name: "courseModulesPassed",
    columns: ("moduleID", "studentID"),
    list(cmp_inserts))

# =====
# 18. WebinarAccess, WebinarAttendance (zwiększone)
# =====
cursor.execute("SELECT webinarID FROM Webinars ORDER BY webinarID")

```

```
for _ in range(NUM_WEBINAR_ACCESS):
    w_id = random.choice(all_webinar_ids)
    s_id = random.choice(student_ids)
    sd = random_datetime( start_year: 2023, end_year: 2025)
    ed = sd + datetime.timedelta(days=30)
    wa_inserts.append((s_id, w_id, sd, ed))
insert_many(cursor, table_name: "WebinarAccess",
            columns: ("studentID", "webinarID", "accessStartDate", "accessEndDate"),
            wa_inserts)

watt_inserts = set()
while len(watt_inserts) < NUM_WEBINAR_ATTENDANCE:
    w_id = random.choice(all_webinar_ids)
    s_id = random.choice(student_ids)
    watt_inserts.add((s_id, w_id))
insert_many(cursor, table_name: "WebinarAttendance",
            columns: ("studentID", "webinarID"),
            list(watt_inserts))

# =====
# 19. studiesParticipants, studyModuleParticipants,
#      extraStudySessionsParticipants
# =====
sp_inserts = set()
while len(sp_inserts) < NUM_STUDIES_PARTICIPANTS:
    sid = random.choice(studies_ids_list)
    stid = random.choice(student_ids)
    sp_inserts.add((sid, stid))
insert_many(cursor, table_name: "studiesParticipants",
            columns: ("studiesID", "studentID"),
            list(sp_inserts))

cursor.execute("SELECT studiesID, moduleID FROM studyModules ORDER BY studiesID, moduleID")
all_study_modules = [(row.studiesID, row.moduleID) for row in cursor.fetchall()]

smp_inserts = set()
while len(smp_inserts) < NUM_STUDY_MODULE_PARTICIPANTS:
    (st_id, mod_id) = random.choice(all_study_modules)
    s_id = random.choice(student_ids)
    smp_inserts.add((s_id, st_id, mod_id))
insert_many(cursor, table_name: "studyModuleParticipants",
            columns: ("studentID", "studiesID", "moduleID"),
```

```

    list(simp_inserts))

cursor.execute("SELECT studiesID, meetingID, moduleID FROM studySessions")
valid_study_sessions = [(row.studiesID, row.meetingID, row.moduleID) for row in cursor.fetchall()]

essp_inserts = set()
desired_count = min(len(valid_study_sessions)//2, 50)
while len(essp_inserts) < desired_count:
    sss = random.choice(valid_study_sessions)
    stid = random.choice(student_ids)
    essp_inserts.add((sss[0], sss[1], sss[2], stid))
insert_many(cursor, table_name: "extraStudySessionsParticipants",
            columns: ("studiesID","meetingID","moduleID","studentID"),
            list(essp_inserts))

# =====
# 20. internship + internshipMeeting
# =====
internship_inserts = []
for _ in range(NUM_INERNSHIP):
    sid = random.choice(studies_ids_list)
    sd = random_date( start_year: 2023, end_year: 2025)
    ed = sd + datetime.timedelta(days=14)
    internship_inserts.append((sid, sd, ed))
insert_many(cursor, table_name: "internship", columns: ("studiesID","startDate","endDate"), internship_inserts)

cursor.execute("SELECT internshipID FROM internship ORDER BY internshipID")
all_internship_ids = [row.internshipID for row in cursor.fetchall()]

im_inserts = []
for _ in range(NUM_INERNSHIP_MEETING):
    if not all_internship_ids:
        break
    i_id = random.choice(all_internship_ids)
    stid = random.choice(student_ids)
    att = random.randint( a: 0, b: 1)
    im_inserts.append((i_id, stid, att))
insert_many(cursor, table_name: "internshipMeeting",
            columns: ("internshipID","studentID","attendance"),
            im_inserts)

# =====

```

```
    # =====#
    # 21. StudentAchievements
    # =====#
    sa_inserts = []
    for _ in range(NUM_STUDENT_ACHIEVEMENTS):
        stid = random.choice(student_ids)
        ach_type = random.choice(["COURSE", "STUDY", "WEBINAR"])
        if ach_type == "COURSE":
            cursor.execute("SELECT courseID FROM Courses")
            all_course_ids2 = [row.courseID for row in cursor.fetchall()]
            val = random.choice(all_course_ids2)
        elif ach_type == "STUDY":
            val = random.choice(studies_ids_list)
        else:
            val = random.choice(all_webinar_ids)
        cert = random.randint( a: 0, b: 1)
        sa_inserts.append((stid, ach_type, val, cert))
    insert_many(cursor, table_name: "StudentAchievements",
                columns: ("studentID", "achievementType", "achievementIDValue", "certificateSent"),
                sa_inserts)

    # ZATWIERDZENIE
    conn.commit()
    conn.close()
    print("Wstawianie danych zakończone sukcesem.")

if __name__ == "__main__":
    main()
```