Kuralay Mukhtar

ID: 210103469

**Exercise 1 + Assignment 2 + Exercise 2 (Go)**

**Report**

* **Connect to a Database:**

This code defines Go structs for each database table, such as Student, Course, Department, Enrollment, and Instructor. These structs represent the schema of the corresponding tables and include fields for primary keys, foreign keys, relationships, and soft deletes. The gorm tags are used to annotate the struct fields with metadata that specifies how they should be mapped to the database.

type Student struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 GPA float64  
 Courses []Course `gorm:"many2many:enrollments"`  
 DepartmentID uint  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

type Course struct {  
 ID uint `gorm:"primaryKey"`  
 Code string  
 Title string  
 Department Department `gorm:"embedded;embeddedPrefix:department\_"`  
 InstructorID uint  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

type Department struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Dean string  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

type Enrollment struct {  
 ID uint `gorm:"primaryKey"`  
 CourseID uint `gorm:"foreignKey:ID"`  
 StudentID uint `gorm:"foreignKey:ID"`  
 Status string  
 Grade string  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

type Instructor struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 Salary int  
 Courses []Course  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

* Use Gorm to establish a connection to the database.

In this code, the database connection is established using Gorm. The ConnectDB() function encapsulates the connection logic, while the main() function demonstrates its usage by calling ConnectDB() and handling any connection errors.

func ConnectDB() (\*gorm.DB, error) {  
 dsn := "host=localhost user=postgres dbname=postgres password=postgres port=5432 sslmode=disable"  
 return gorm.Open(postgres.Open(dsn), &gorm.Config{})  
}

func main() {  
 db, err := ConnectDB()  
 if err != nil {  
 panic("failed to connect database: " + err.Error())  
 }

**CRUD Operations:**

* Implement CRUD operations for each table (Create, Read, Update, Delete).
* AddNewStudent: This function adds a new student to the database. It utilizes the `Create` method of the `gorm.DB` object to insert the student, returning any encountered error.
* RetrieveStudentByID: Given a student ID, this function retrieves the corresponding student from the database using the `First` method. It returns the student along with any encountered error, such as a failure to find the student or a database query error.

- UpdateStudent: This function updates an existing student's record in the database. It employs the `Save` method of the `gorm.DB` object to update the student's information, returning any encountered error.

* DeleteStudentByID: For a given student ID, this function deletes the corresponding student from the database using the `Delete` method. It returns any encountered error, such as a failure to delete the student or a database operation error.

Similarly for the other tables, there are corresponding CRUD operations implemented to handle the creation, retrieval, updating, and deletion of records in the database.

func AddNewStudent(db \*gorm.DB, student \*Student) error {  
 return db.Create(student).Error  
}  
  
func RetrieveStudentByID(db \*gorm.DB, id uint) (\*Student, error) {  
 var student Student  
 if err := db.First(&student, id).Error; err != nil {  
 return nil, err  
 }  
 return &student, nil  
}  
  
func UpdateStudent(db \*gorm.DB, student \*Student) error {  
 return db.Save(student).Error  
}  
  
func DeleteStudentByID(db \*gorm.DB, id uint) error {  
 return db.Delete(&Student{}, id).Error  
}

func AddNewCourse(db \*gorm.DB, course \*Course) error {  
 return db.Create(course).Error  
}  
  
func RetrieveCourseByID(db \*gorm.DB, id uint) (\*Course, error) {  
 var course Course  
 if err := db.First(&course, id).Error; err != nil {  
 return nil, err  
 }  
 return &course, nil  
}  
  
func UpdateCourse(db \*gorm.DB, course \*Course) error {  
 return db.Save(course).Error  
}  
  
func DeleteCourseByID(db \*gorm.DB, id uint) error {  
 return db.Delete(&Course{}, id).Error  
}

func AddNewDepartment(db \*gorm.DB, department \*Department) error {  
 return db.Create(department).Error  
}  
  
func RetrieveDepartmentByID(db \*gorm.DB, id uint) (\*Department, error) {  
 var department Department  
 if err := db.First(&department, id).Error; err != nil {  
 return nil, err  
 }  
 return &department, nil  
}  
  
func UpdateDepartment(db \*gorm.DB, department \*Department) error {  
 return db.Save(department).Error  
}  
  
func DeleteDepartmentByID(db \*gorm.DB, id uint) error {  
 return db.Delete(&Department{}, id).Error  
}

func AddNewEnrollment(db \*gorm.DB, enrollment \*Enrollment) error {  
 return db.Create(enrollment).Error  
}  
  
func RetrieveEnrollmentByID(db \*gorm.DB, id uint) (\*Enrollment, error) {  
 var enrollment Enrollment  
 if err := db.First(&enrollment, id).Error; err != nil {  
 return nil, err  
 }  
 return &enrollment, nil  
}  
  
func UpdateEnrollment(db \*gorm.DB, enrollment \*Enrollment) error {  
 return db.Save(enrollment).Error  
}  
  
func DeleteEnrollmentByID(db \*gorm.DB, id uint) error {  
 return db.Delete(&Enrollment{}, id).Error  
}

func AddNewInstructor(db \*gorm.DB, instructor \*Instructor) error {  
 return db.Create(instructor).Error  
}  
  
func RetrieveInstructorByID(db \*gorm.DB, id uint) (\*Instructor, error) {  
 var instructor Instructor  
 if err := db.First(&instructor, id).Error; err != nil {  
 return nil, err  
 }  
 return &instructor, nil  
}  
  
func UpdateInstructor(db \*gorm.DB, instructor \*Instructor) error {  
 return db.Save(instructor).Error  
}  
  
func DeleteInstructorByID(db \*gorm.DB, id uint) error {  
 return db.Delete(&Instructor{}, id).Error  
}

* For example, create functions to add a new student, retrieve a course, update an instructor, and delete a department.

Create: This segment demonstrates the creation of a new student named "Nurkhat Muratkhan" with a GPA of 3.4 using the `AddNewStudent` function. If an error occurs during the addition process, a panic message is triggered with details of the error. Upon successful addition, a confirmation message is printed.

Read: Here, a course with the ID of 1 is fetched from the database using the `RetrieveCourseByID` function. If the retrieval fails, a panic message with the corresponding error is triggered. Otherwise, the retrieved course information is displayed.

Update: An instructor with the ID of 3 is retrieved from the database, and their salary is updated to 600000. The `UpdateInstructor` function is employed for this update operation. If any error occurs during the update process, a panic message with the error details is displayed. Upon successful update, a confirmation message is printed.

Delete: This part depicts the deletion of a department with the ID of 3 using the `DeleteDepartmentByID` function. If an error occurs during the deletion process, a panic message with the error details is triggered. Upon successful deletion, a confirmation message is printed.

//CREATE  
student := &Student{Name: "Nurkhat", Surname: "Muratkhan", GPA: 3.4}  
if err := AddNewStudent(db, student); err != nil {  
 panic(fmt.Errorf("failed to add student: %v", err))  
}  
fmt.Println("New student added successfully")  
  
//READ  
courseID := uint(1)  
course, err := RetrieveCourseByID(db, courseID)  
if err != nil {  
 panic("failed to retrieve course: " + err.Error())  
}  
fmt.Println("Retrieved course:", course)  
  
//UPDATE  
instructorID := uint(3)  
instructor, err := RetrieveInstructorByID(db, instructorID)  
if err != nil {  
 panic("failed to retrieve instructor: " + err.Error())  
}  
  
instructor.Salary = 600000  
err = UpdateInstructor(db, instructor)  
if err != nil {  
 panic("failed to update instructor: " + err.Error())  
}  
fmt.Println("Instructor updated successfully")  
  
//DELETE  
departmentID := uint(3)  
err = DeleteDepartmentByID(db, departmentID)  
if err != nil {  
 panic("failed to delete department: " + err.Error())  
}  
fmt.Println("Department deleted successfully")

**Migrations:**

* Create Gorm migrations for each table to set up their initial schema.

This function, MigrateDB, is responsible for setting up the initial schema for each table, including Department, Course, Instructor, Student, and Enrollment. It utilizes the AutoMigrate method provided by Gorm to automatically create or update the tables based on the defined structs. If any error occurs during the migration process, it is returned.

* Update the schema of one of the tables (e.g., add a new column) and run the migration to reflect the changes in the database.

Additionally, the code executes a SQL statement to add a new column named "age" to the "students" table using the ALTER TABLE command. This step demonstrates updating the schema of one of the tables to reflect changes in the database structure. If an error occurs during the execution of the SQL statement, it is returned.

func MigrateDB(db \*gorm.DB) error {  
 err := db.AutoMigrate(&Department{}, &Course{}, &Instructor{}, &Student{}, &Enrollment{})  
 if err != nil {  
 return err  
 }  
  
 err = db.Exec("ALTER TABLE students ADD COLUMN age INT").Error  
 if err != nil {  
 return err  
 }  
  
 return nil  
}

err = MigrateDB(db)  
if err != nil {  
 panic("failed to migrate database: " + err.Error())  
}

**Querying:**

* Implement functions to query the database for each table.

This code defines a function named GetRowCount to retrieve the number of rows in a specified table within the database. It accepts a pointer to the Gorm database instance (db) and the name of the table (tableName) as parameters. The function utilizes Gorm's Model, Table, and Count methods to execute a query and count the number of records in the specified table. If any error occurs during the query execution, it is returned along with a count of zero.

func GetRowCount(db \*gorm.DB, tableName string) (int64, error) {  
 var count int64  
 if err := db.Model(&gorm.Model{}).Table(tableName).Count(&count).Error; err != nil {  
 return 0, err  
 }  
 return count, nil  
}

Within the main code, the GetRowCount function is invoked for each table (departments, courses, instructors, students, and enrollments) to retrieve the respective row counts. If an error occurs during any of these queries, a panic message with the error details is triggered.

departmentsCount, err := GetRowCount(db, "departments")  
if err != nil {  
 panic("failed to get row count for departments: " + err.Error())  
}  
  
coursesCount, err := GetRowCount(db, "courses")  
if err != nil {  
 panic("failed to get row count for courses: " + err.Error())  
}  
  
instructorsCount, err := GetRowCount(db, "instructors")  
if err != nil {  
 panic("failed to get row count for instructors: " + err.Error())  
}  
  
studentsCount, err := GetRowCount(db, "students")  
if err != nil {  
 panic("failed to get row count for students: " + err.Error())  
}  
  
enrollmentsCount, err := GetRowCount(db, "enrollments")  
if err != nil {  
 panic("failed to get row count for enrollments: " + err.Error())  
}

fmt.Printf("Number of rows in departments table: %d\n", departmentsCount)  
fmt.Printf("Number of rows in courses table: %d\n", coursesCount)  
fmt.Printf("Number of rows in instructors table: %d\n", instructorsCount)  
fmt.Printf("Number of rows in students table: %d\n", studentsCount)  
fmt.Printf("Number of rows in enrollments table: %d\n", enrollmentsCount)

* Retrieve all students from a specific department.

The provided code illustrates a function named GetStudentsByDepartment designed to fetch all students belonging to a particular department from the database. It accepts two parameters: the database instance (db) and the name of the department (departmentName).

Inside the function, a query is constructed using GORM's chaining methods to join multiple tables (students, enrollments, courses, and departments) based on their relationships. The Where clause filters the results to include only students associated with the specified department.

func GetStudentsByDepartment(db \*gorm.DB, departmentName string) ([]Student, error) {  
 var students []Student  
 if err := db.Table("students").Joins("JOIN enrollments ON students.id = enrollments.student\_id").  
 Joins("JOIN courses ON enrollments.course\_id = courses.id").  
 Joins("JOIN departments ON courses.department\_id = departments.id").  
 Where("departments.name = ?", departmentName).  
 Find(&students).Error; err != nil {  
 return nil, err  
 }  
 return students, nil  
}

In the main code, the GetStudentsByDepartment function is called with the department name as an argument. If an error occurs during the retrieval process, an error message is printed. Otherwise, the retrieved students' information, including their ID, name, surname, and GPA, is displayed iteratively.

students, err := GetStudentsByDepartment(db, "Computer Science")  
if err != nil {  
 fmt.Println("Failed to retrieve students:", err)  
 return  
}  
  
fmt.Println("Students in Computer Science department:")  
for \_, student := range students {  
 fmt.Printf("ID: %d, Name: %s, Surname: %s, GPA: %.2f\n", student.ID, student.Name, student.Surname, student.GPA)  
}

* Find courses taught by a particular instructor.

This function, GetCoursesByInstructor, retrieves courses taught by a specific instructor from the database. It accepts the database instance (db) and the instructor's name (instructorName) as parameters. Using GORM's query chaining methods, it constructs a query to join the courses and instructors tables based on their relationship. The function then filters the results to include only courses taught by the specified instructor. Finally, it returns the list of courses along with any potential error.

func GetCoursesByInstructor(db \*gorm.DB, instructorName string) ([]Course, error) {  
 var courses []Course  
 if err := db.Table("courses").Joins("JOIN instructors ON courses.instructor\_id = instructors.id").  
 Where("instructors.name = ?", instructorName).  
 Find(&courses).Error; err != nil {  
 return nil, err  
 }  
 return courses, nil  
}

In the main code, after invoking this function with the instructor's name, it prints the details of the retrieved courses, such as their ID, code, and title.

courseList, err := GetCoursesByInstructor(db, "John Doe")  
if err != nil {  
 fmt.Println("Failed to retrieve courses:", err)  
 return  
}  
  
fmt.Println("Courses taught by John Doe:")  
for \_, course := range courseList {  
 fmt.Printf("ID: %d, Code: %s, Title: %s\n", course.ID, course.Code, course.Title)  
}

* Get all enrollments for a specific student.

The GetEnrollmentsByStudent function retrieves all enrollments associated with a particular student from the database. It takes the database instance (db) and the student's ID (studentID) as parameters. Using GORM's querying capabilities, it filters enrollments based on the provided student ID. The function then returns the list of enrollments along with any potential error. In the main code, after calling this function with the student's ID, it displays the details of each enrollment, including its ID, course ID, student ID, status, and grade.

func GetEnrollmentsByStudent(db \*gorm.DB, studentID uint) ([]Enrollment, error) {  
 var enrollments []Enrollment  
 if err := db.Where("student\_id = ?", studentID).Find(&enrollments).Error; err != nil {  
 return nil, err  
 }  
 return enrollments, nil  
}

enrollments, err := GetEnrollmentsByStudent(db, 103)  
if err != nil {  
 fmt.Println("Failed to retrieve enrollments:", err)  
 return  
}  
  
fmt.Println("Enrollments for student with ID 123:")  
for \_, enrollment := range enrollments {  
 fmt.Printf("ID: %d, CourseID: %d, StudentID: %d, Status: %s, Grade: %s\n", enrollment.ID, enrollment.CourseID, enrollment.StudentID, enrollment.Status, enrollment.Grade)  
}

**Associations:**

* Define relationships between tables.
* For example, set up a many-to-many relationship between Students and Courses through the Enrollments table.

This relationship is established through the enrollments table, which acts as a join table linking students to courses.

type Student struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 GPA float64  
 Courses []Course `gorm:"many2many:enrollments"`  
 DepartmentID uint  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

* Establish a one-to-many relationship between Instructors and Courses.

Each course is associated with a single instructor identified by their ID. The Instructor struct includes a slice of Course objects representing the courses taught by each instructor.

type Course struct {  
 ID uint `gorm:"primaryKey"`  
 Code string  
 Title string  
 Department Department `gorm:"embedded;embeddedPrefix:department\_"`  
 InstructorID uint  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

type Instructor struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 Salary int  
 Courses []Course  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

**Transactions:**

* Implement a transaction that enrolls a student in a course. Ensure that both the student and course records are updated within the same transaction.

This code demonstrates the implementation of a transaction to enroll a student in a course. The EnrollStudentInCourse function accepts three parameters: the database instance (db), the ID of the student to be enrolled (studentID), and the ID of the course in which the student will be enrolled (courseID). Within the function, a transaction (tx) is initiated using db.Begin(). Then, an Enrollment struct is created with the provided studentID, courseID, and a status of "Enrolled". This enrollment record is then created within the transaction using tx.Create(&enrollment). If any error occurs during the creation of the enrollment record, the transaction is rolled back using tx.Rollback(), and the error is returned. Otherwise, if the enrollment is successful, the transaction is committed using tx.Commit().

func EnrollStudentInCourse(db \*gorm.DB, studentID, courseID uint) error {  
 tx := db.Begin()  
  
 enrollment := Enrollment{StudentID: studentID, CourseID: courseID, Status: "Enrolled"}  
 if err := tx.Create(&enrollment).Error; err != nil {  
 tx.Rollback()  
 return fmt.Errorf("failed to enroll student in course: %w", err)  
 }  
  
 tx.Commit()  
  
 return nil  
}

In the main code, the EnrollStudentInCourse function is called with the student ID and course ID as arguments. If an error occurs during the enrollment process, a panic message is triggered with details of the error. Otherwise, a success message indicating that the student has been enrolled in the course is printed to the console.

//TRANSACTION  
err = EnrollStudentInCourse(db, 104, 2)  
if err != nil {  
 panic("failed to enroll student in course: " + err.Error())  
}  
  
fmt.Println("Student enrolled in course successfully!")

**Hooks:**

* Use Gorm hooks to automatically update timestamps when a student is created or an instructor is updated.

This code demonstrates the implementation of Gorm hooks to automatically update timestamps when a student is created or an instructor is updated. For the Student struct, a BeforeCreate hook is defined, which sets the CreatedAt and UpdatedAt timestamps to the current time when a new student is created. Similarly, for the Instructor struct, a BeforeUpdate hook is implemented to update the UpdatedAt timestamp to the current time when an existing instructor's information is updated.

type Student struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 GPA float64  
 Courses []Course `gorm:"many2many:enrollments"`  
 DepartmentID uint  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

func (s \*Student) BeforeCreate(tx \*gorm.DB) (err error) {  
 s.CreatedAt = time.Now()  
 s.UpdatedAt = time.Now()  
 return nil  
}

func (i \*Instructor) BeforeUpdate(tx \*gorm.DB) (err error) {  
 i.UpdatedAt = time.Now()  
 return nil  
}

type Instructor struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 Salary int  
 Courses []Course  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

Additionally, the code includes examples of creating a new student and updating an existing instructor to validate the functionality of the implemented hooks.

student2 := &Student{ID: 109, Name: "John", Surname: "Doe", GPA: 3.5}  
if err := db.Create(student2).Error; err != nil {  
 panic(fmt.Errorf("failed to create student: %v", err))  
}  
fmt.Println("New student created successfully")  
  
instructor2 := &Instructor{ID: 1, Name: "Jane", Surname: "Smith", Salary: 50000}  
if err := db.Save(instructor2).Error; err != nil {  
 panic(fmt.Errorf("failed to update instructor: %v", err))  
}  
fmt.Println("Instructor updated successfully")





**Soft Deletes:**

* Implement soft deletes for one of the tables (e.g., Students). Add a "deleted\_at" column and modify the delete operation to set this column.

The Student struct includes a DeletedAt field tagged with gorm:"index", indicating soft deletes.

type Student struct {  
 ID uint `gorm:"primaryKey"`  
 Name string  
 Surname string  
 GPA float64  
 Courses []Course `gorm:"many2many:enrollments"`  
 DepartmentID uint  
 CreatedAt time.Time  
 UpdatedAt time.Time  
 DeletedAt \*gorm.DeletedAt `gorm:"index"`  
}

The SoftDeleteStudentByID function finds and soft deletes a student by their ID, setting the DeletedAt field.

func SoftDeleteStudentByID(db \*gorm.DB, studentID uint) error {  
 var student Student  
 if err := db.First(&student, studentID).Error; err != nil {  
 return fmt.Errorf("failed to find student: %v", err)  
 }  
  
 if err := db.Delete(&student).Error; err != nil {  
 return fmt.Errorf("failed to soft delete student: %v", err)  
 }  
  
 return nil  
}

Usage involves calling SoftDeleteStudentByID with the student's ID. If successful, a confirmation message is printed.

studentID := uint(106)  
if err := SoftDeleteStudentByID(db, studentID); err != nil {  
 panic(err)  
}  
  
fmt.Println("Student soft deleted successfully")



**Custom Queries:**

* Write custom queries to get specific information from the database.
* For example, find the total number of students in a department, get the list of courses with the number of enrolled students, etc.

GetStudentCountByDepartment: This function retrieves the total number of students in a department by querying the Student table based on the provided department ID. It returns the count of students and any error encountered during the process.

func GetStudentCountByDepartment(db \*gorm.DB, departmentID uint) (int64, error) {  
 var count int64  
 if err := db.Model(&Student{}).  
 Where("department\_id = ?", departmentID).  
 Count(&count).Error; err != nil {  
 return 0, err  
 }  
 return count, nil  
}

departmentID2 := uint(1)  
studentCount, err := GetStudentCountByDepartment(db, departmentID2)  
if err != nil {  
 panic("failed to get student count by department: " + err.Error())  
}  
department, err := RetrieveDepartmentByID(db, departmentID2)  
if err != nil {  
 panic("failed to retrieve department: " + err.Error())  
}  
fmt.Printf("Total number of students in department '%s': %d\n", department.Name, studentCount)

GetCoursesWithEnrollmentCount: This function retrieves a list of courses along with the count of enrolled students for each course. It utilizes a custom SQL query to join the Course and Enrollment tables and groups the results by course title. The function returns a map containing course titles as keys and their respective enrollment counts as values.

func GetCoursesWithEnrollmentCount(db \*gorm.DB) (map[string]int, error) {  
 var courses []struct {  
 Title string  
 EnrollmentCount int  
 }  
 if err := db.Model(&Course{}).  
 Select("courses.title, COUNT(enrollments.id) as enrollment\_count").  
 Joins("LEFT JOIN enrollments ON courses.id = enrollments.course\_id").  
 Group("courses.id, courses.title").  
 Find(&courses).Error; err != nil {  
 return nil, err  
 }  
  
 courseEnrollmentCounts := make(map[string]int)  
 for \_, course := range courses {  
 courseEnrollmentCounts[course.Title] = course.EnrollmentCount  
 }  
 return courseEnrollmentCounts, nil  
}

courses, err := GetCoursesWithEnrollmentCount(db)  
if err != nil {  
 panic("failed to get courses with enrollment count: " + err.Error())  
}  
fmt.Println("Courses with enrollment count:")  
for title, enrollmentCount := range courses {  
 fmt.Printf("Course: %s, Enrollment Count: %d\n", title, enrollmentCount)  
}

**Testing:**

* Write unit tests for your Gorm functions. Use an in-memory database or a test database to ensure that your functions work as expected without affecting the production database.

The provided unit test for the GetStudentCountByDepartment function validates its functionality by setting up an in-memory SQLite database, populating it with test data, invoking the function under test, and verifying that the returned student count matches the expected value for a given department ID. The test ensures correct behavior and error handling, contributing to the reliability of the function.

func TestGetStudentCountByDepartment(t \*testing.T) {  
 db, err := gorm.Open(sqlite.Open("file::memory:?cache=shared"), &gorm.Config{})  
 if err != nil {  
 t.Fatalf("failed to open database: %v", err)  
 }  
  
 db.AutoMigrate(&Department{})  
 db.Create(&Department{ID: 1, Name: "Test Department"})  
  
 db.AutoMigrate(&Student{})  
 db.Create(&Student{ID: 9, Name: "Test Student", Surname: "Test", DepartmentID: 3})  
  
 count, err := GetStudentCountByDepartment(db, 3)  
 if err != nil {  
 t.Fatalf("failed to get student count by department: %v", err)  
 }  
  
 expected := int64(3)  
 if count != expected {  
 t.Errorf("got %d students in 'Test Department', expected %d", count, expected)  
 }  
}