EF Core Advanced Lab:
University Course Management System

You are developing a University Course Management System where multiple entities interact, such as Course, Student, Instructor, Department, and Enrolment. The system must handle complex relationships between these entities, support inheritance for different types of courses, and ensure performance optimization.

Key Skills: Incorporating key EF Core features like one-to-one, one-to-many, many-to-many relationships, inheritance strategies, and performance tips:

Part 1: Database Design and Relationships

Task 1.1: One-to-One Relationship

- **Entity Design**: Implement a one-to-one relationship between Instructor and OfficeAssignment. Every instructor has an office, and every office is assigned to one instructor.
- **Requirements**:
  - Create Instructor and OfficeAssignment entities.
  - Configure the one-to-one relationship using EF Core Fluent API.
  - Add data annotations where appropriate.

Questions:

  - How do you configure the one-to-one relationship using the Fluent API?
  - What considerations would you take to ensure the integrity of this relationship when deleting an instructor?

Task 1.2: One-to-Many Relationship

- **Entity Design**: Create a one-to-many relationship between Department and Course. A department can offer multiple courses, but each course belongs to only one department.
- **Requirements**:
  - Define Department and Course entities.
  - Configure the one-to-many relationship.
  - Implement a feature to cascade delete courses when a department is deleted.

Questions:

  - How do you handle cascading deletes in EF Core, and what impact does it have on performance?

    - What are the pros and cons of using data annotations versus the Fluent API for relationship configuration?

Task 1.3: Many-to-Many Relationship

- **Entity Design**: Create a many-to-many relationship between Course and Student through an Enrollment entity. A student can enroll in multiple courses, and each course can have many students.
- **Requirements**:
    - Define Course, Student, and Enrollment entities.
    - Configure the many-to-many relationship.
    - Implement logic to calculate the average grade for each course based on the Enrollment data.

Questions:

- How would you configure the many-to-many relationship using EF Core 5.0 or later?
- Discuss the performance implications of many-to-many relationships and how to optimize queries in this context.

Part 2: Inheritance Strategies

Task 2.1: Table Per Hierarchy (TPH)

- **Scenario**: The university offers different types of courses, such as OnlineCourse and OnSiteCourse. Use TPH to implement inheritance for the Course entity.
- **Requirements**:
    - Create a base Course entity and derive OnlineCourse and OnSiteCourse from it.
    - Configure the TPH inheritance strategy.
    - Implement logic to distinguish between online and onsite courses when querying the database.

Questions:

- How does the TPH strategy affect database design and querying in EF Core?
- What are the pros and cons of using TPH compared to other inheritance strategies like TPT or TPC?

Task 2.2: Table Per Type (TPT)

- **Scenario**: Suppose the university decides to maintain separate tables for OnlineCourse and OnSiteCourse. Use TPT to implement this scenario.
- **Requirements**:
    - Redefine the OnlineCourse and OnSiteCourse entities using the TPT strategy.

- Migrate the existing TPH implementation to TPT without data loss.
- Analyze the performance impact of the TPT strategy compared to TPH.

Questions:

- What are the key differences between TPT and TPH regarding query performance and database design?
- How would you manage the migration process from TPH to TPT in a production environment?

Part 3: Performance Optimization

Task 3.1: Query Optimization

- **Scenario**: The system needs to handle large datasets, such as a student count of over 100,000. Optimize the following query for fetching courses along with their enrolled students:

```
var courses = context.Courses
                    .Include(c => c.Enrollments)
                    .ThenInclude(e => e.Student)
                    .ToList();
```

- **Requirements**:
    - Identify performance bottlenecks in the above query.
    - Optimize the query using EF Core features like AsNoTracking, Split Queries, or Filtered Includes.
    - Measure the performance difference before and after optimization.

Questions:

- How does AsNoTracking impact query performance and when should it be used?
- What are Split Queries, and how do they improve performance in EF Core?

Task 3.2: Batch Operations

- **Scenario**: The university needs to update the course fee for all OnlineCourse instances. Implement an efficient batch update operation.
- **Requirements**:
    - Write an EF Core query to update all OnlineCourse records in a single database operation.
    - Compare the performance of the batch operation with an iterative update approach.
    - Implement logging to track the performance of the update operation.

Questions:

- What are the benefits and potential risks of batch operations in EF Core?
- How does EF Core handle concurrency in batch updates?

Submission Instructions

1. Submit your EF Core codebase as a GitHub repository.
2. Provide a detailed readme file explaining your design decisions, especially regarding relationship configurations, inheritance strategies, and performance optimizations.
3. Include unit tests for critical parts of the implementation.
4. Discuss any challenges you encountered and how you addressed them.

This assessment will evaluate your understanding of EF Core's advanced features and your ability to apply them in real-world scenarios.