Lab Exercise: Practicing the Database-First Approach in EF Core

Objective:

This lab exercise will guide you through the process of using the Database-First approach in Entity Framework Core. You'll reverse-engineer an existing database to generate entity classes and a DbContext in a .NET Core project. The exercise involves working with a university database schema containing several tables.

Prerequisites:

- Basic understanding of Entity Framework Core.
- A .NET Core development environment installed (e.g., Visual Studio, Visual Studio Code, or JetBrains Rider).
- SQL Server or another supported database system installed and running.
- Familiarity with SQL and database management.

Database Schema:

The database schema consists of the following tables:

1. **Students**
    - StudentID (Primary Key, int)
    - FirstName (nvarchar(50))
    - LastName (nvarchar(50))
    - EnrollmentDate (datetime)
    - Email (nvarchar(100))
2. **Courses**
    - CourseID (Primary Key, int)
    - Title (nvarchar(100))
    - Credits (int)
    - DepartmentID (Foreign Key, int)
3. **Enrollments**
    - EnrollmentID (Primary Key, int)
    - CourseID (Foreign Key, int)
    - StudentID (Foreign Key, int)
    - Grade (int)

4. **Departments**
    - DepartmentID (Primary Key, int)
    - Name (nvarchar(100))
    - Budget (decimal)

Steps:

Part 1: Setting Up the Database

1. **Create the Database**:
    - Use SQL Server Management Studio (SSMS) or another database tool to create a new database named UniversityDB.
    - Execute the following SQL scripts to create the necessary tables:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    EnrollmentDate DATETIME,
    Email NVARCHAR(100)
);

CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Budget DECIMAL(18, 2)
);

CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    Title NVARCHAR(100),
    Credits INT,
    DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID)
);

CREATE TABLE Enrollments (
```

```
    EnrollmentID INT PRIMARY KEY,
    CourseID INT FOREIGN KEY REFERENCES Courses(CourseID),
    StudentID INT FOREIGN KEY REFERENCES Students(StudentID),
    Grade INT
);
```

2. **Populate the Tables**:
    - Insert sample data into each table for testing. For example:

```
INSERT INTO Departments (DepartmentID, Name, Budget) VALUES (1, 'Computer Science', 50000);

INSERT INTO Courses (CourseID, Title, Credits, DepartmentID) VALUES (1, 'Database Systems', 4, 1);

INSERT INTO Students (StudentID, FirstName, LastName, EnrollmentDate, Email) VALUES (1, 'John', 'Doe', '2023-01-10', 'john.doe@example.com');

INSERT INTO Enrollments (EnrollmentID, CourseID, StudentID, Grade) VALUES (1, 1, 1, 95);
```

Part 2: Reverse Engineering with EF Core

1. **Create a New .NET Core Project**:
    - Open your preferred development environment and create a new console application project named UniversityApp.
2. **Install EF Core Tools**:
    - Install the necessary EF Core packages by running the following command in the Package Manager Console or via the CLI:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Design
```

3. **Reverse Engineer the Database**:
    - Use the following command to scaffold the database:

```
dotnet ef dbcontext scaffold "Server=your_server_name;Database=UniversityDB;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Models
```

    - This command will generate entity classes and a DbContext class named UniversityDbContext in a folder named Models.
4. **Review the Generated Code**:
    - Explore the Models folder to review the generated entity classes (Student, Course, Enrollment, Department) and the UniversityDbContext class.

- Note how the relationships between the entities are represented by navigation properties.

Part 3: Working with the DbContext

1. **Querying Data**:
    - Write a console application that uses the UniversityDbContext to query data from the database. For example, fetch and display all students and their enrollments:

```
using (var context = new UniversityDbContext())
{
    var students = context.Students
                          .Include(s => s.Enrollments)
                          .ThenInclude(e => e.Course)
                          .ToList();

    foreach (var student in students)
    {
        Console.WriteLine($"{student.FirstName} {student.LastName}");
        foreach (var enrollment in student.Enrollments)
        {
            Console.WriteLine($"  Enrolled in: 
{enrollment.Course.Title}, Grade: {enrollment.Grade}");
        }
    }
}
```

2. **Adding and Updating Data**:
    - Add a new student to the database and enroll them in a course:

```
using (var context = new UniversityDbContext())
{
    var newStudent = new Student
    {
        FirstName = "Jane",
        LastName = "Smith",
        EnrollmentDate = DateTime.Now,
        Email = "jane.smith@example.com"
```

```
    };

    context.Students.Add(newStudent);
    context.SaveChanges();

    var enrollment = new Enrollment
    {
        StudentID = newStudent.StudentID,
        CourseID = 1, // Assuming course ID 1 exists
        Grade = 90
    };

    context.Enrollments.Add(enrollment);
    context.SaveChanges();
}
```

3. **Deleting Data**:
    - Write code to delete a student and their related enrollments from the database:

```
using (var context = new UniversityDbContext())
{
    var student = context.Students
                         .Include(s => s.Enrollments)
                         .FirstOrDefault(s => s.StudentID == 1);

    if (student != null)
    {
        context.Students.Remove(student);
        context.SaveChanges();
    }
}
```

Part 4: Conclusion and Best Practices

- **Discussion**:
    - Reflect on the differences between the Database-First and Code-First approaches.

    - Discuss scenarios where the Database-First approach might be more suitable.
    - Consider performance implications of using the Database-First approach, especially in large, complex databases.
- **Best Practices**:
    - Always validate and customize the generated code to fit the specific needs of your application.
    - Use partial classes to extend generated classes without modifying the scaffolded code.
    - Regularly update the model if the database schema changes.

Submission:

- Submit the complete code for the console application as a GitHub repository.
- Include a readme file explaining your experience with the Database-First approach and any challenges you faced.

This lab exercise will give you hands-on experience with the Database-First approach in EF Core, providing a solid foundation for working with existing databases in .NET applications.