# Manging Migrations - EF Core - Exercise

Managing migrations exercise for the [Databases Advanced - Entity Framework course @ SoftUni](https://softuni.bg/trainings/4540/entity-framework-core-june-2024).

# Welcome to Managing Migrations with EF Core

In this exercise, you will learn how to create, manage, and apply migrations to a database using EF Core. This is an essential skill for any developer working with data-driven applications, as it allows you to keep your database schema in sync with your application's data model.

## What You Will Learn

### Database First Approach

We will **start with an existing database** and **generate the corresponding EF Core model classes**

### Creating Migrations

You will learn **how to create migrations to update the database schema** when the model changes

### Applying Migrations

We will cover **how to apply migrations to your database** to keep it up-to-date

### Seeding Database

Learn how to **seed the database with initial data** during the migration process

### Handling Errors

As with any development process, **you may encounter errors**. We will learn **how to troubleshoot** and   
**resolve common issues** that arise during migrations

# Practical Guide

## Database First Approach

### Save the Script

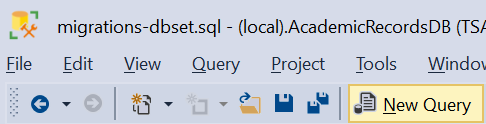
Save the provided SQL script as **migrations-dbset.sql**

### SQL Server Management Studio (SSMS) A logo of a tool and a cylinder A screenshot of a computer Description automatically generated

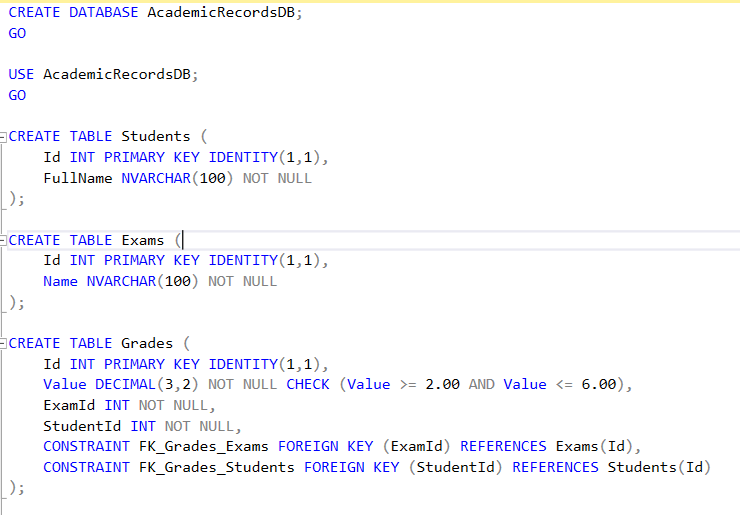
Open SSMS and Connect to your SQL Server instance

### Create a New Query

Open a **New Query** window



Copy and paste the contents of the **migrations-dbset.sql** script into the query window



### Execute the Script

Execute the script to create the **AcademicRecordsDB** database, the schema, and populate it with sample data

A screenshot of a computer

Description automatically generated

### Simple Database Diagram

Now we have managed to create our **database in it's initial state**



## Creating a CS Project and Scaffolding Database

### Set Up the Project in Visual Studio

**Launch Visual Studio** and select **Create a new project**

A screenshot of a computer screen

Description automatically generated

Create New **Console App** (.NET Core) -> Name your project (e.g., **AcademicRecordsApp**) and click **Create**

A screenshot of a computer

Description automatically generated A screen shot of a computer

Description automatically generated

### Add EF Core Packages

**Right-click** on the **project in Solution Explorer** and select **Manage NuGet Packages**

**A screenshot of a computer program

Description automatically generated** **A screenshot of a computer

Description automatically generated**

### Run the Scaffold Command

In the **Package Manager Console**, run the following command to scaffold the database

If the **SQL Server instance is running on the local machine**, the server name might be **localhost** or '**.**'

Scaffold-DbContext "Server=YOUR\_SERVER\_NAME;Database=AcademicRecordsDB; Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

This command will **generate entity classes** and a **context class** based on the **existing database schema**

### Verify the Scaffolded Code

After running the scaffold command, a **Models** folder will be **created (or updated)** in your project directory

A screenshot of a computer

Description automatically generated A screen shot of a computer

Description automatically generated

This folder will contain the **entity classes representing your database tables** (**Students**, **Exams**, **Grades**) as well as the **context class** (**AcademicRecordsDBContext**)

Open the **generated entity classes** and the **context class** to **familiarize yourself with the   
structure and relationships** defined

### The Process of Scaffolding

**Scaffolding** and **migrations** are **both essential aspects of working with Entity Framework Core**, but they serve **different purposes** and are **used in different contexts**

Scaffolding is the **process of generating code based on an existing database schema**. It is used to **create entity classes and a DbContext class that correspond to the tables and relationships** in your database

You **use scaffolding** when you have an **existing database** and you want to **generate the initial model code** to start working with EF Core in your application

### Differences Between Scaffolding and Migrations

Starting Point:

* **Scaffolding**: Starts with an existing database schema and generates the corresponding model code
* **Migrations**: Starts with an existing model and generates code to update the database schema

Purpose:

* **Scaffolding**: Database-first approach (from database to code)
* **Migrations**: Code-first or model-first approach (from code to database)

Why Scaffolding is Not a Migration:

* **Scaffolding** is a **one-time process** to generate the initial code based on an existing database.  
   It **does not track changes or apply incremental updates**
* **Migrations** are used to **manage and apply incremental updates** to the database schema over time.   
  They **track changes made to the model and generate scripts to apply those changes to the database**

### Organize the Project Structure and Rebuild the Solution

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Detailed Analysis of AcademicRecordsDBContext

### Namespace and Using Directives

The namespace is **AcademicRecordsApp.Data**, which aligns with the new folder structure

It includes **necessary directives for EF Core** and collections

A screenshot of a computer

Description automatically generated

### Constructors

The **parameterless constructor** is **useful for scenarios where options are set externally**

The constructor with **DbContextOptions<AcademicRecordsDBContext>** **allows passing configuration options** when instantiating the context

A screenshot of a computer program

Description automatically generated

### DbSet Properties

A group of text on a white background

Description automatically generated

These properties **represent the tables in the database** and **provide a way to query and save instances**   
of these entities

### OnConfiguring Method

This method is **called during context configuration**, it **sets up the connection string for the SQL Server database**.

A computer screen shot of a computer code

Description automatically generated

The **#warning** directive **suggests moving the connection string out of the source code** for security reasons

### OnModelCreating Method

**Configures the model's properties** and **relationships using the Fluent API**

**Defines constraints and relationships** for the **Exam**, **Grade**, and **Student** entities

A screen shot of a computer code

Description automatically generated

### Partial Method



This partial method **allows for further customization in another partial class**

## Evolving Database and Models

This **section will guide you through the process of evolving your database schema and models**. You will learn how to **add new entities**, **modify existing ones**, and **apply these changes to the database using EF Core migrations**

### Adding a New Entity: Course

**Add a new class** called **Course.cs** in the Models folder

A screenshot of a computer

Description automatically generated A computer screen shot of a code

Description automatically generated

**Update the Student entity class. Modify the Student class to include a collection of Courses**

**A computer screen shot of a program

Description automatically generated**

### Update the DBContext

**Add a DbSet for the Course entity** in **AcademicRecordsDBContext.cs**

A screenshot of a computer code

Description automatically generated

A close-up of a computer code

Description automatically generated

**Set the maximum length** for the Name property to **100 characters. HasMaxLength()** is a method that specifies the maximum length for a string property in the database

### Creating and Applying the Migration

**Ensure the Model Changes**:

* Created **Course.cs**
* Updated **Student.cs**
* Updated **AcademicRecordsDBContext.cs**

**Add a Migration**

* Open the **Package Manager Console** (Go to Tools > NuGet Package Manager > Package Manager ConsoleA screenshot of a computer

  Description automatically generated
* Run the **Add-Migration** Command

**A screenshot of a computer

Description automatically generated**

* **Review the Migration Script**

Visual Studio will **generate a migration script**. Review the script to understand what changes will be applied to the database

**A close-up of a computer code

Description automatically generated** **A screen shot of a computer

Description automatically generated**

### Update-Database

Run the Update-Database Command in the Package Manager Console

A screenshot of a computer

Description automatically generated

### Encounter Update-Database Errors

A screenshot of a computer

Description automatically generated

The migration is **creating the Exams, Students, and other tables** because **it doesn't have any record of the current database schema**. This can happen if the **migration history is out of sync with the actual database schema**, or if the **initial migration didn't accurately reflect the existing state of the database**

### Sync Migration History with the Database

**Remove Existing Migrations**

* **Delete all existing migration files in the Migrations folder**

A screenshot of a computer

Description automatically generated

**Reset the Migration History**

* **If necessary, you can manually reset the migration history table in the database. This is typically the \_\_EFMigrationsHistory table**
* Open SQL Server Management Studio (SSMS) and run the following command to clear the migration history

A screenshot of a computer

Description automatically generated

**Create an Initial Baseline Migration**

* Create a new initial migration that reflects the current state of the database without trying to create tables that already exist

**A screenshot of a computer

Description automatically generated**

**Manually Adjust the Initial Migration**

* Open the generated migration file and **remove or comment out the creation of existing tables**

**A screen shot of a computer code

Description automatically generated**

**A computer code with green text

Description automatically generated**

**A computer screen shot of a program

Description automatically generated**

**Ensure that the Down method does not drop the existing tables (Exams, Students, Grades). You should only drop the tables that were added in this migration (Courses and CourseStudent).**

**A computer screen shot of a program

Description automatically generated**

### Rename Using Search and Replace

Yes, it is generally better to have the join table named **StudentsCourses** instead of **CourseStudent**, following the **convention of placing the primary entity name first**, which helps in **readability and consistency**.

* Press **Ctrl + Shift + H** to open the **"Replace in Files"** dialog

A screenshot of a computer

Description automatically generated A close-up of a text

Description automatically generated

### Update Database

A close up of numbers

Description automatically generated A screenshot of a computer

Description automatically generated

### Error Explanation

**Initial Error:**

* We **created the Course entity** and **updated the Student entity** to include a collection of Courses **without properly configuring the relationships** in the **OnModelCreating** method
* As a result, **the generated migration script attempted to recreate existing tables** (Exams, Students, Grades), leading to conflicts and errors

**Lack of Fluent API Configuration:**

* **Without using the Fluent API to configure the many-to-many relationship**, EF Core was **unable to correctly create and manage the join table** for the Students and Courses relationship
* This **caused issues with the migration process** and could lead to an inconsistent or incorrect database schema
* The **Fluent API allows** you to explicitly **define the relationships between entities**, ensuring that EF Core correctly understands and manages the relationships
* **Proper configuration helps avoid conflicts and errors during the migration process**, ensuring that existing tables and data are not inadvertently modified or deleted

**Correct Configuration with Fluent API:**

* To correct the mistake, the OnModelCreating method should be updated in the **AcademicRecordsDBContext** to explicitly define the **many-to-many relationship** between **Students** and **Courses** using the **Fluent API**

A computer code with text

Description automatically generated

### Resolve the Issue

**Remove Existing Migrations**

* **Delete all existing migration** **files** and **reset the migration history** in the database to ensure a clean state

**Create an Initial Baseline Migration**

* Let's start from the beginning by **scaffolding the database again** and then **creating an initial migration**

A computer screen shot of a computer screen

Description automatically generated

**Creating Course Entity && Configure Relationships Using Fluent API**



**Apply the Migration**

**A screenshot of a computer program

Description automatically generated**

### Updated Database Diagram

**Refresh** the existing tables, and **build a New Diagram**

A screenshot of a computer

Description automatically generated

In the **updated database schema**, we have introduced **two significant changes**. First, **we added a new table** called Courses to store information about the various courses offered, each identified by a unique Id and a Name. Second, we established a **many-to-many relationship between Students and Courses** using a join table named **StudentsCourses**. This join table includes foreign keys StudentsId and CoursesId, linking students to the courses they are enrolled in.

## Creating New Data Model Relations

In this section, we will focus on **establishing a new relationship between Student, Course and Exam entities**. Specifically, we will define that **every exam belongs to one course only**, while **one course can have many exams**. This relationship is known as a **one-to-many relationship**. Additionally, we will establish a **many-to-many relationship between Student and Course**, where **every student can enroll in multiple courses**, and **each course can have many students**. Follow these steps to create and configure these relationships in your data model.

### Using Data Anotations to Configure Relationships

**Data Annotations** provide a **way to configure the model classes directly using attributes**. This can make the model classes more readable by keeping the configuration close to the properties they configure.

### Update Entities

**Exam Entity:**

* In the Exam entity, we have introduced several enhancements **using Data Annotations** to ensure proper **configuration and validation of the database schema**.
* We added a **CourseId** property to **serve as a foreign key**, which establishes a **one-to-many relationship** with the **Course** entity. This is enforced by the **[ForeignKey("CourseId")]** annotation.
* Additionally, we used the **[Required]** and **[MaxLength(100)]** annotations on the Name property to ensure that every exam has a name and that it does not exceed 100 characters in length.
* The Course **navigation property** provides a **link to the associated Course entity**, allowing us to navigate from an Exam to the Course it belongs to.
* These changes ensure that each exam is associated with a specific course, enhancing data integrity and the logical structure of our database.



A screenshot of a computer program

Description automatically generated

**Course Entity:**

* In the **Course** entity, we have implemented several enhancements **using Data Annotations** to ensure **accurate configuration and validation of the database schema**
* We introduced a **Name** property with the **[Required]** and **[MaxLength(100)]** annotations to guarantee that each course has a name and that the name does not exceed 100 characters in length
* The **Exams** **navigation property** was added to represent the collection of Exam entities associated with a particular course. This creates a **one-to-many relationship**, allowing each course to have multiple exams
* These changes enhance the relational structure of the database, ensuring that courses can be linked to their respective exams, thereby improving data organization and integrity. The Course entity now accurately reflects the academic structure, where each course can encompass various exams



A screen shot of a computer code

Description automatically generated

**Student Entity:**

* In the Student entity, we have introduced several enhancements **using Data Annotations** to ensure proper **configuration and validation of the database schema**
* We added a FullName property with the **[Required]** and **[MaxLength(100)]** annotations to ensure that **every student has a full name and that it does not exceed 100 characters in length**
* The **Grades navigation property** was added to represent the collection of Grade entities associated with a particular student. This allows **each student to have multiple grades for different exams**
* The **Courses navigation property** was **added to establish a many-to-many relationship between Student and Course entities**. This allows **each student to enroll in multiple courses**, and **each course to have multiple students**
* These changes enhance the relational structure of the database, ensuring that students can be linked to their respective grades and courses, thereby improving data organization and integrity. The Student entity now accurately reflects the academic structure, where each student can have multiple grades and enroll in multiple courses



A screen shot of a computer

Description automatically generated

### Update DBContext

In this section, we will **configure the relationships and constraints in our DbContext using the Fluent API**. This configuration ensures that our entity relationships are accurately represented in the database schema.

**Configuring Relationships in DbContext**

We will update the **OnModelCreating** method in the **DbContext** to **define the relationships** between the **Student**, **Course**, and **Exam** entities **using the Fluent API**

****

### Add the Migration and Update the Database

A close-up of a computer screen

Description automatically generated

* We have successfully added the new migration called **StudentExamCourse\_RelationsAdded**
* We are going to attempt to **update the database with this new migration**

A screenshot of a computer

Description automatically generated

* We **attempted to update the database with the new migration**, but **encountered an error**
* The error message indicates that **there was a conflict with the FOREIGN KEY constraint**

### Understanding the Issue

The error occurs because the **Exams table has a foreign key constraint that references the Courses table**. The specific issue is likely **caused by existing data in the Exams table** that **doesn't match any Id in the Courses table**

**This can happen if:**

* **There are records in the Exams table with CourseId values that don't exist in the Courses table**
* **The CourseId column in the Exams table has NULL values, and the foreign key constraint doesn't allow NULLs**

### Resolve the Problem

**Remove the Failed Migration:**

* Since the **update was unsuccessful**, we'll first **remove** the failed **migration**

A computer screen shot of a number

Description automatically generated

* **Modify OnModelCreating to Allow Nulls Temporarily**

**A computer code with text

Description automatically generated**

### What Will Happen Later When the Property is Made Required?

* **Schema Update**
  + EF Core will generate a migration that updates the schema to enforce the property as required (non-nullable)
  + This will alter the table definition to disallow nulls in the column
* **Data Validation**
  + Existing data must comply with the new non-nullable constraint. If there are any null values in the column, the migration will fail
  + EF Core will ensure at the application level that any new records or updates to this column must provide a non-null value
* **Foreign Key Constraints**
  + The foreign key relationship will enforce that every entry in the Exams table must reference a valid Id in the Courses table
  + This ensures data integrity by preventing orphaned records in the Exams table

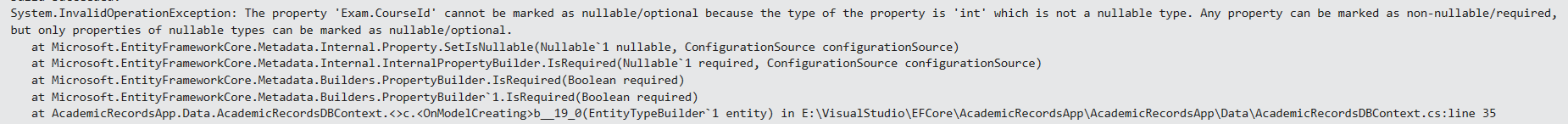
### Add and Apply a Migration

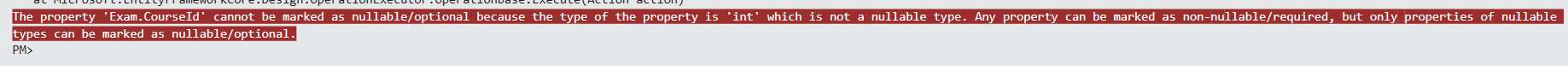
**It seems that the migration will be successful this time**

A screenshot of a computer

Description automatically generated

**But encounter a new issue**





We encountered an **error while attempting to make the CourseId property nullable**. The error message indicates that **the property cannot be marked as nullable unless it is defined as a nullable type in the model class**

### Nullable Integer Property Resolution

**Remove the Failed Migration**

* **Remove** the failed **migration** to **revert the model snapshot**

**Update the Model Class**

* **Remove the Required Attribute**
* Modify the Exam model to make the **CourseId** property a **nullable integer (int?)**

A computer screen shot of a computer code

Description automatically generated

**Add a New Migration and Update the Database**

A computer screen shot of a computer program

Description automatically generated

### Completing the Migration and Updating the Database Schema

Congratulations on successfully creating the migration and updating the database schema!

