

EF Core Migrations

Sync Db Schemas and Applications



SoftUni Team
Technical Trainers



SoftUni

Software University

<https://about.softuni.bg/>

sli.do

#csharp-db

Table of Contents

1. Migrations Overview
2. Applying Migrations
3. Understanding the **Connection String**
4. Managing Migrations
5. Customize **Migration Code**
6. EF Core **Migration Commands**
7. Multiple **Providers**
8. Error Handling





Migrations Overview

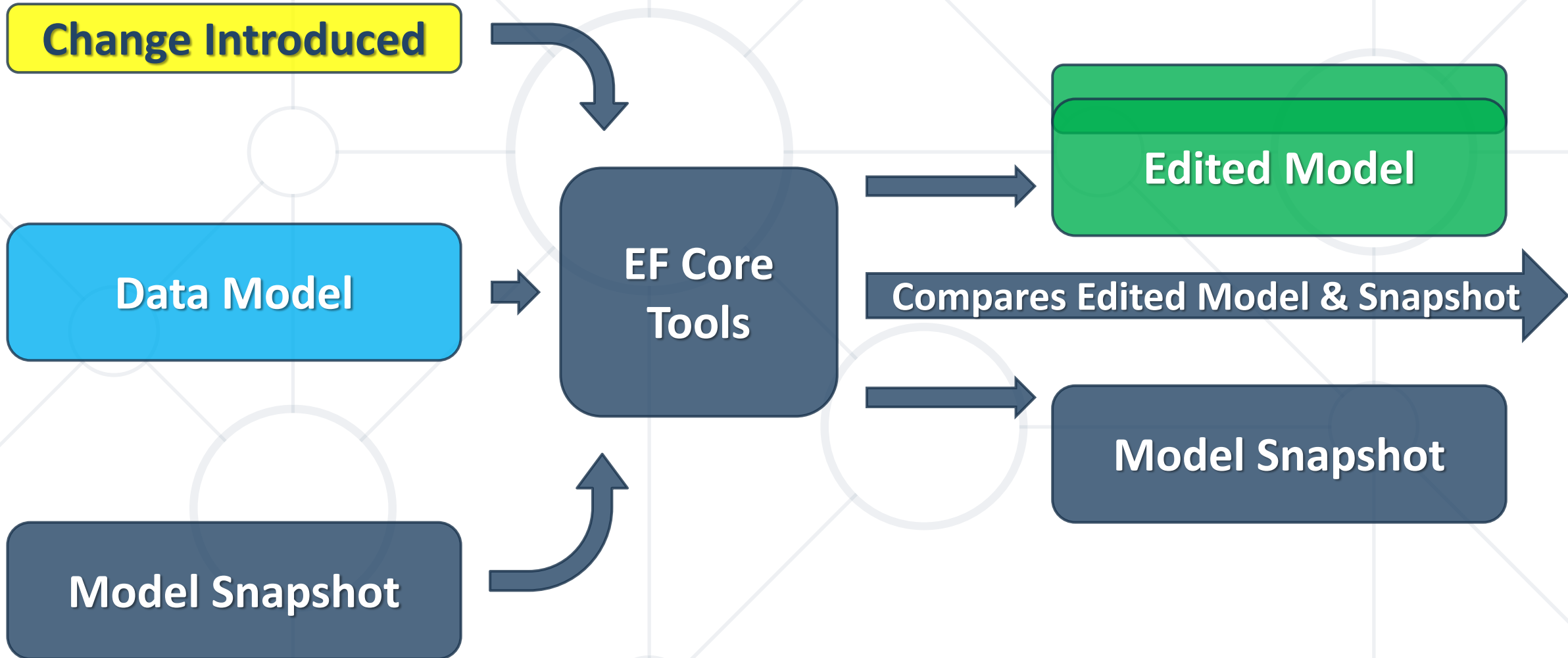
First Steps with Migrations

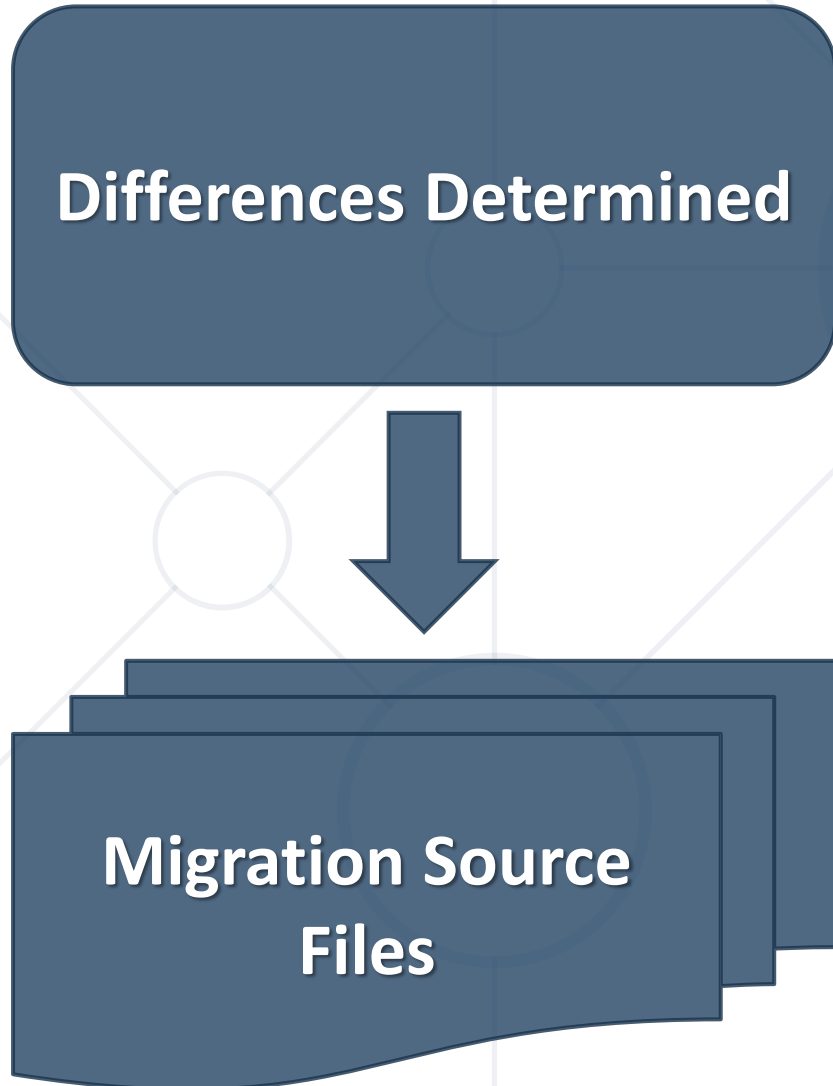
Migrations in EF Core



- **Data models change** as features get implemented
 - Add **new entities** or **properties**
 - Remove entities, properties
- Migrations feature in EF Core:
 - Allows to **update** the **db schema**
 - Keeps the db schema in **sync with the data model**
 - Protects the **existing data** in the database

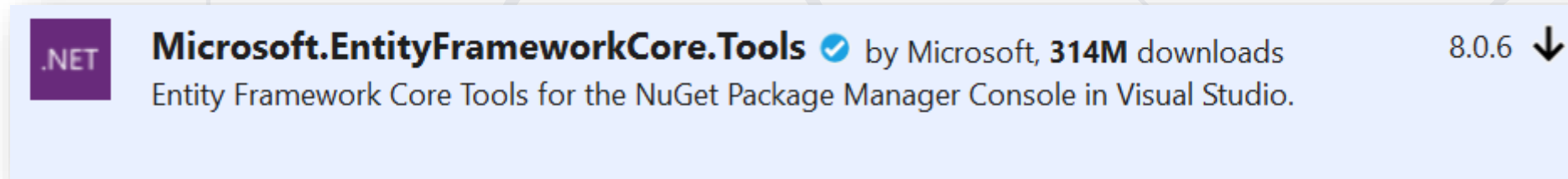
How do Migrations Function





- **Migrations Source Files**
 - Can be tracked in the project's **source control** like any other source file
 - Once a **Migration** has been **generated**, it can be **applied** to a database in various ways
 - EF Core **records all applied migrations** in a **history table**

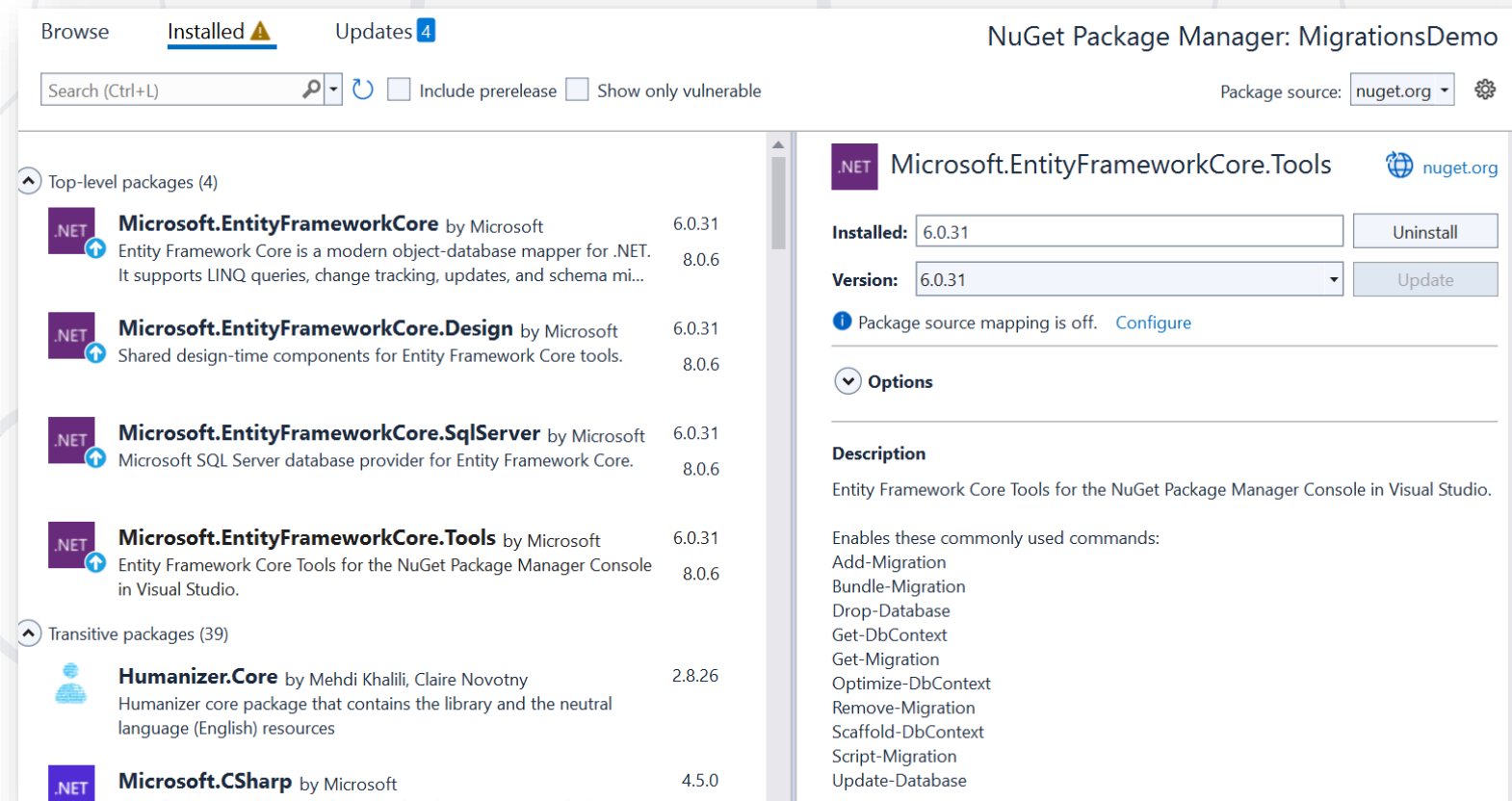
- Manage **NuGet Packages** —→ EntityFrameworkCore.Tools



- Create a conceptual **model from an existing database**
 - Then graphically visualize and edit your conceptual model
- Graphically create a conceptual **model first**
 - Then generate a database that supports your model

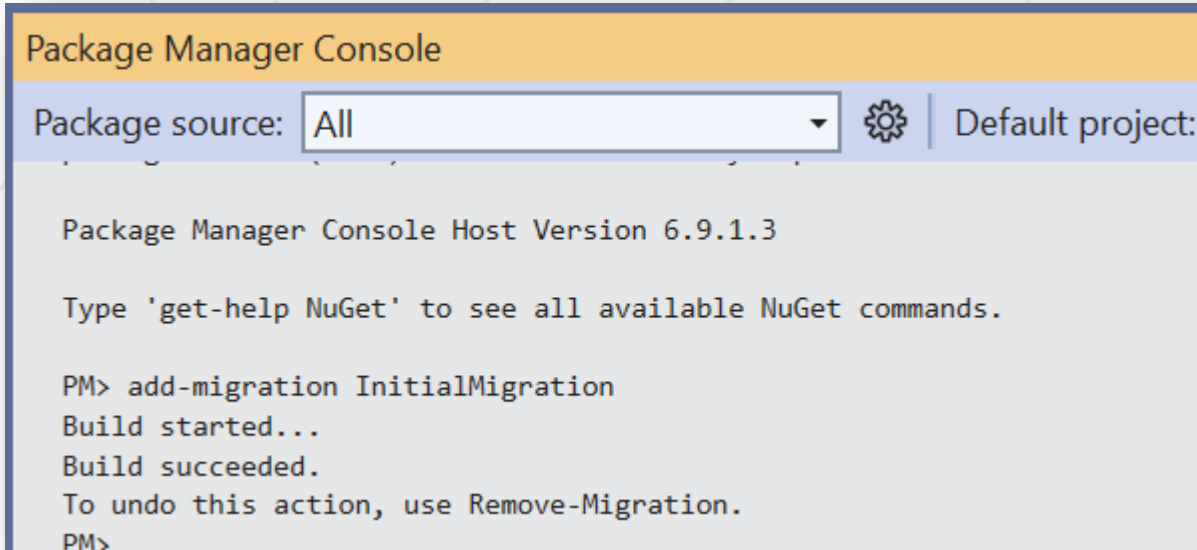
Create Migration – NuGet Packages

- Start the **MigrationsDemo** project in Visual Studio
- EFCore.Tools is added to your **MigrationsDemo** project

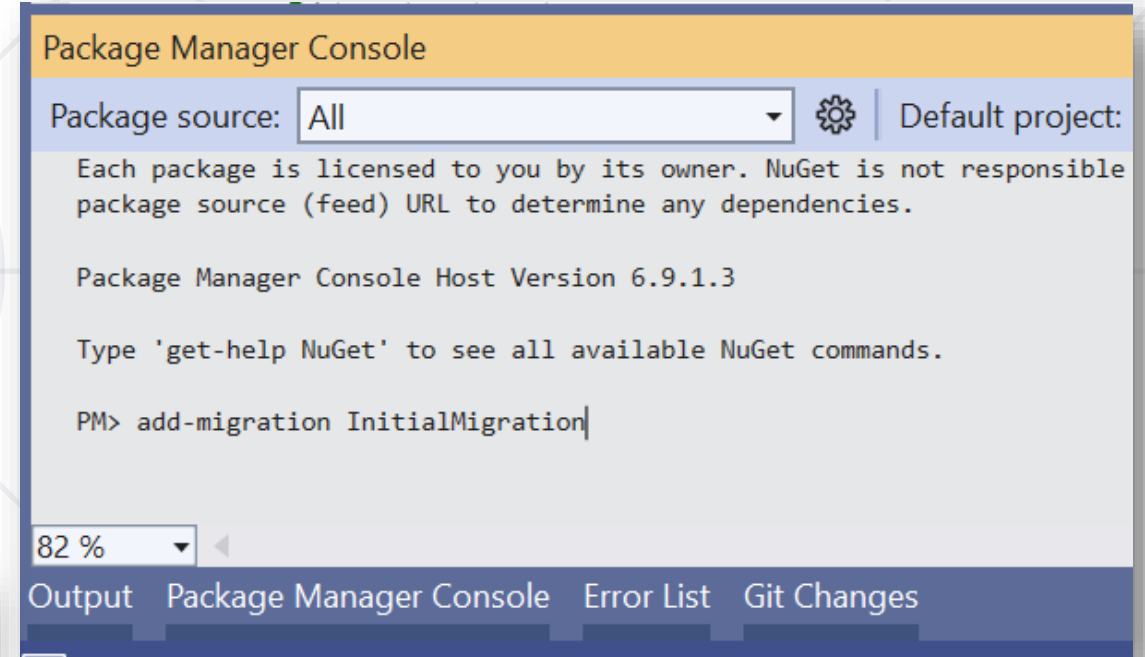


Create Migration – Package Manager Console

- Add a migration to the project
- Run the command shown
 - A migration will be **created** and **applied**



```
Package Manager Console
Package source: All [v] [gear] Default project:
Package Manager Console Host Version 6.9.1.3
Type 'get-help NuGet' to see all available NuGet commands.
PM> add-migration InitialMigration
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

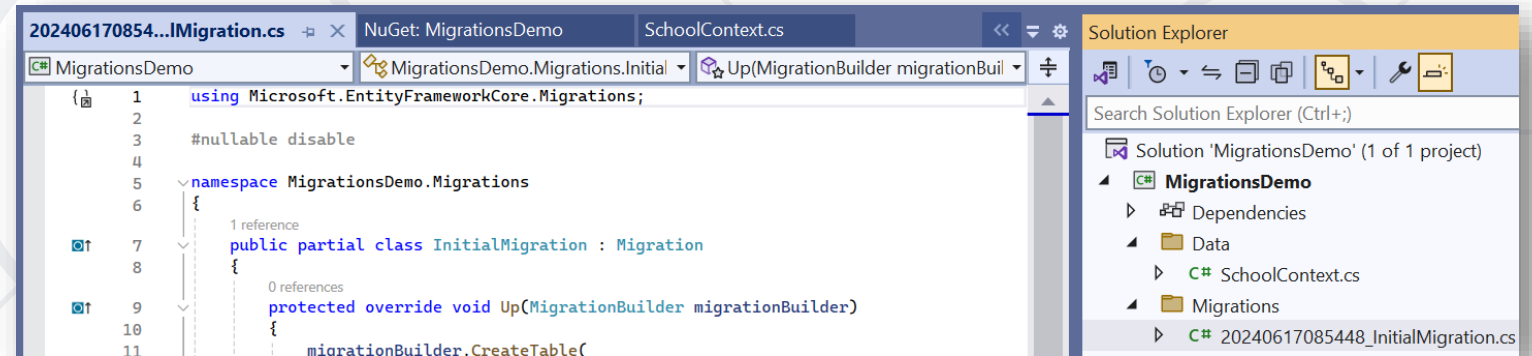


```
Package Manager Console
Package source: All [v] [gear] Default project:
Each package is licensed to you by its owner. NuGet is not responsible
package source (feed) URL to determine any dependencies.
Package Manager Console Host Version 6.9.1.3
Type 'get-help NuGet' to see all available NuGet commands.
PM> add-migration InitialMigration|
82 % [v] [left arrow]
Output Package Manager Console Error List Git Changes
```

Create Migration – Migration Source Files

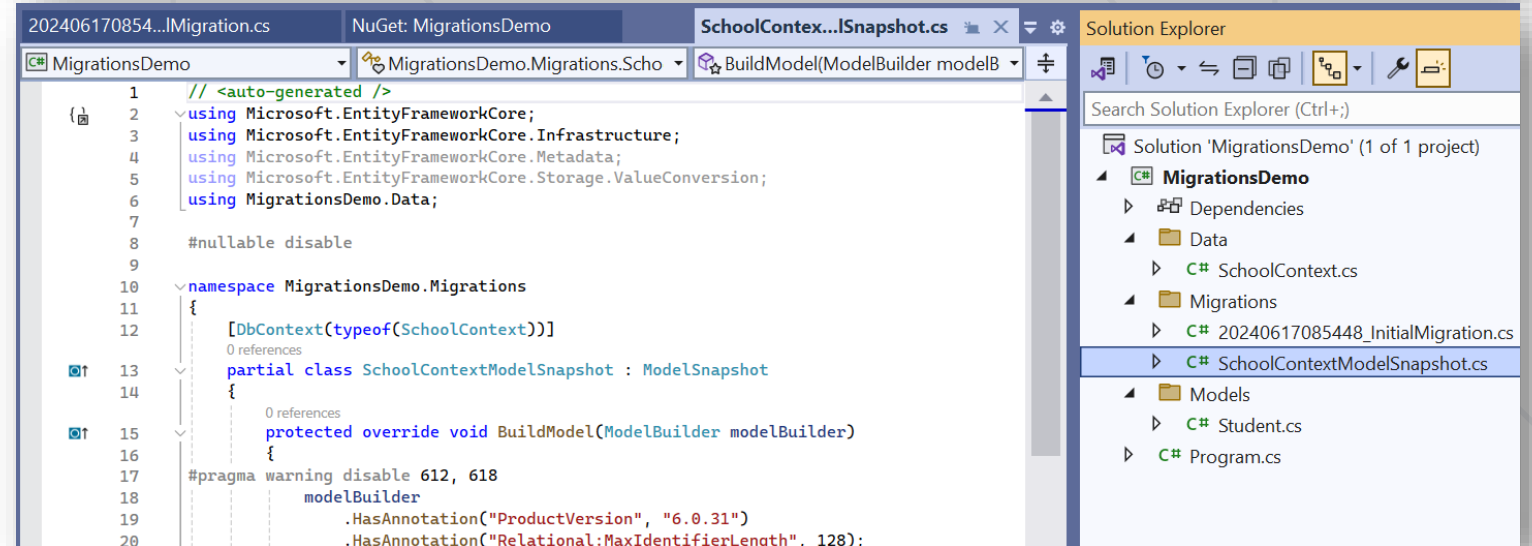
- Each Migration typically generates two main files:

- Migration Class File



```
1 using Microsoft.EntityFrameworkCore.Migrations;
2
3 #nullable disable
4
5 namespace MigrationsDemo.Migrations
6 {
7     1 reference
8     public partial class InitialMigration : Migration
9     {
10         0 references
11         protected override void Up(MigrationBuilder migrationBuilder)
12         {
13             migrationBuilder.CreateTable(
```

- Model Snapshot



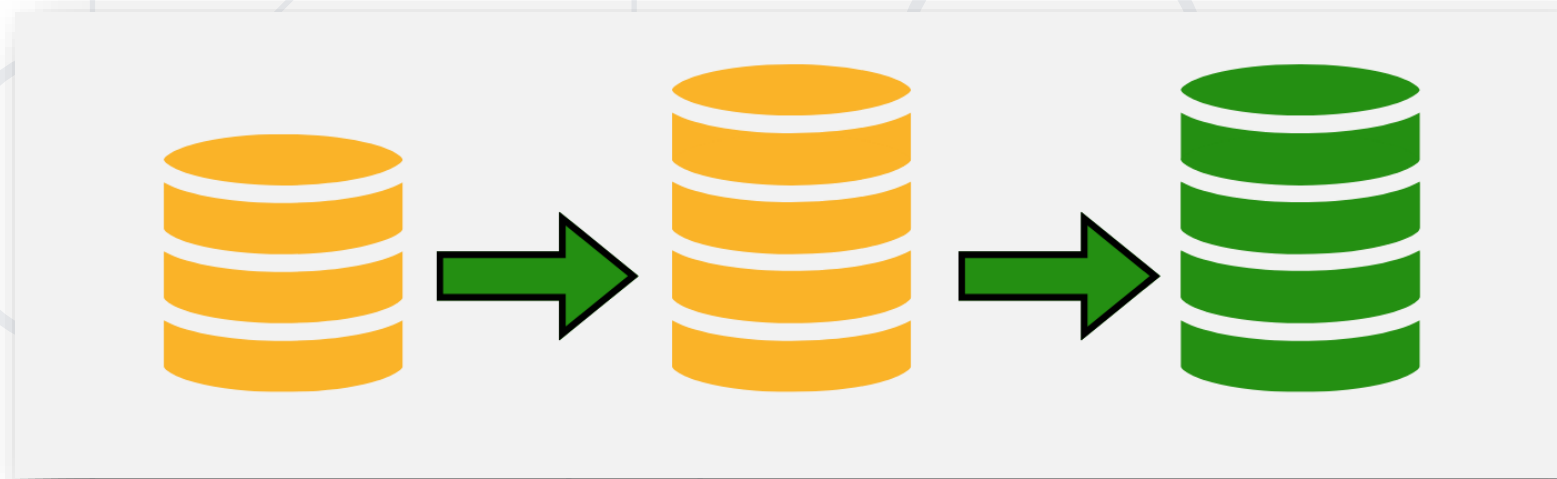
```
1 // <auto-generated />
2 using Microsoft.EntityFrameworkCore;
3 using Microsoft.EntityFrameworkCore.Infrastructure;
4 using Microsoft.EntityFrameworkCore.Metadata;
5 using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
6 using MigrationsDemo.Data;
7
8 #nullable disable
9
10 namespace MigrationsDemo.Migrations
11 {
12     [DbContext(typeof(SchoolContext))]
13     0 references
14     partial class SchoolContextModelSnapshot : ModelSnapshot
15     {
16         0 references
17         protected override void BuildModel(ModelBuilder modelBuilder)
18         {
19             #pragma warning disable 612, 618
20             modelBuilder
21                 .HasAnnotation("ProductVersion", "6.0.31")
22                 .HasAnnotation("Relational:MaxIdentifierLength", 128);
```



Applying Migrations

Database Update

- What is **'update-database'** ?
 - The **'update-database'** command in Entity Framework Core **applies any pending migrations** to the database
 - It ensures that the **database schema** matches the **current state** of the **EF Core model**



How It Works?

- Checks Migration History
 - Compares the **applied migrations** (recorded in '**__EFMigrationsHistory**' table) with **available migrations**
- Generates SQL
 - **Creates SQL scripts** based on the '**Up**' methods in pending migrations
- Executes SQL
 - **Runs the SQL scripts** to update the database schema
- Updates History
 - Records the applied migrations in the '**__EFMigrationsHistory**' table to **avoid reapplying** them



Understanding the Connection String

- A **Connection String** is a string that specifies information about a data source and the means of connecting to it
- It includes details such as the **database server**, **database name**, **authentication credentials**, and other configuration settings...

Standard Security

```
Server=myServerAddress; Database=myDataBase; User Id=myUsername; Password=myPassword;
```

Db Server

Db Name

Credentials

Connection String Components

- **Server**
 - The **name or network address** of the **SQL Server** instance (e.g., (localhost)\\mssqllocaldb)
- **Database**
 - The **name of the database** to connect to (e.g., School)
- **Authentication Details**
 - Specify the **use of Authentication**
- **Other settings** to **communicate** with the db server



Configuring the Connection String

- The **Connection String** could be defined in the 'OnConfiguring' method of the **DbContext** class
- Example:

```
public class SchoolContext : DbContext
{
    0 references
    public DbSet<Student> Students { get; set; } = null!;

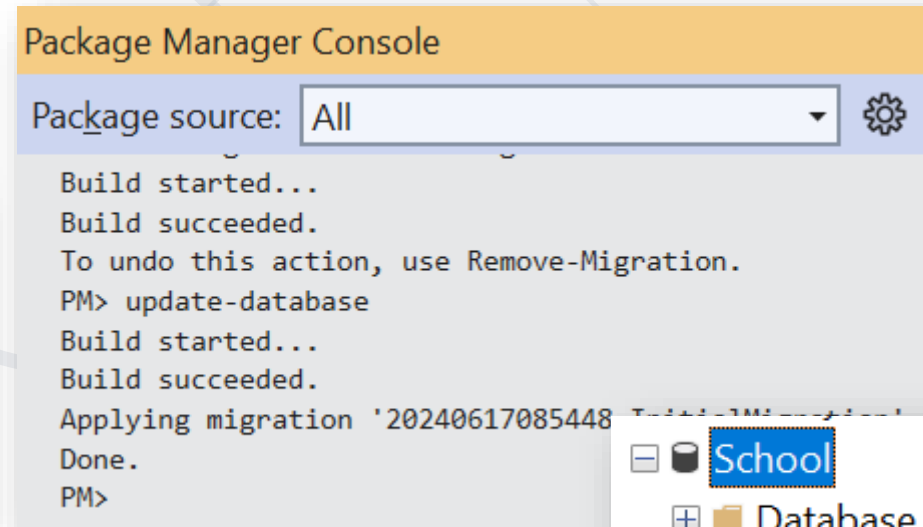
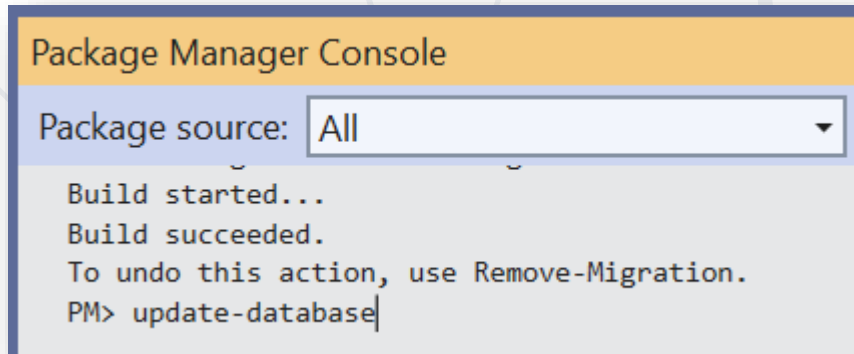
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=localhost;Database=School;Trusted_Connection=True;");
    }
}
```

Trusted Connection

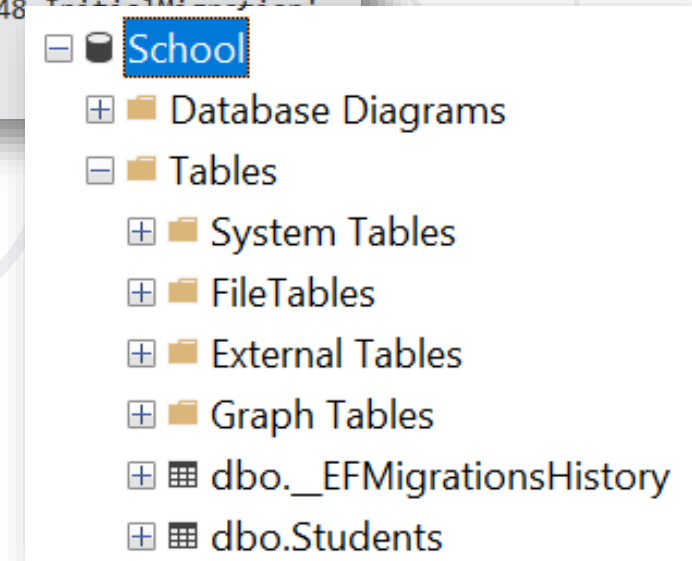
```
Server=myServerAddress; Database=myDataBase; Trusted_Connection=True;
```

Running the 'Update-Database' Command

- Type the '**update-database**' command (case insensitive)
- Press '**Enter**'



- This will **apply any pending migrations** to the database





Managing Migrations

Evolving the Models

Evolving the Model

- Evolve the **Student** model by adding a new property:
 - Open the Student class file
- Add a **new property** Email

```
public class Student
{
    0 references
    public int Id { get; set; }

    0 references
    public string Name { get; set; } = null!;

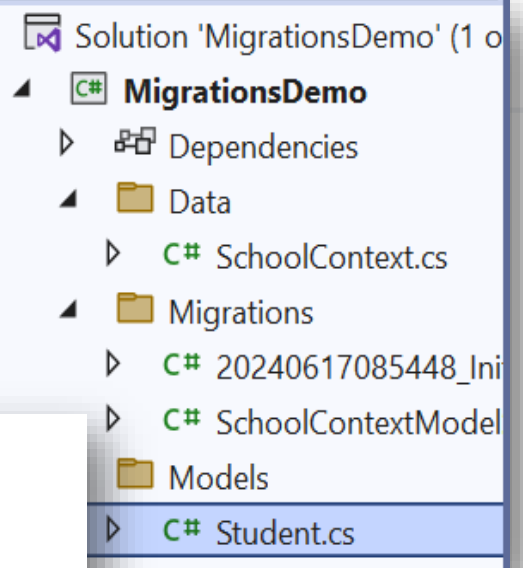
    0 references
    public int Age { get; set; }
}

public class Student
{
    0 references
    public int Id { get; set; }

    0 references
    public string Name { get; set; } = null!;

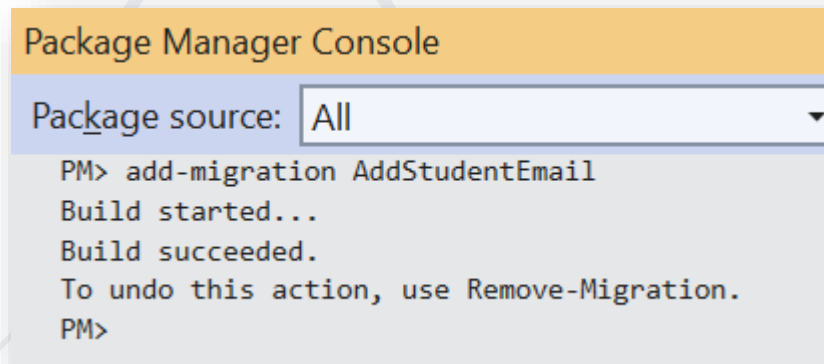
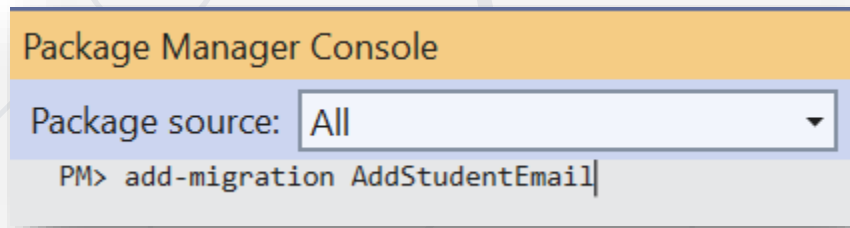
    0 references
    public int Age { get; set; }

    0 references
    public string Email { get; set; } = null!;
}
```

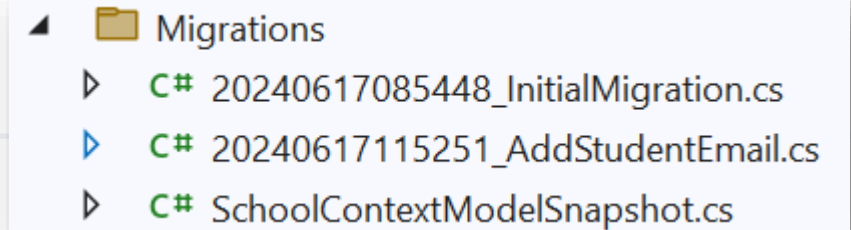


Creating a New Migration

- Open the Package Manager Console in Visual Studio
- Run the command to create a new migration



- **EF Core** generates a new migration file in the '**Migrations**' folder
- **Up Method**
 - Adds the Email column to the 'Students' table
- **Down Method**
 - Removes the Email column if the migration is **rolled back**



```
namespace MigrationsDemo.Migrations
{
    1 reference
    public partial class AddStudentEmail : Migration
    {
        0 references
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "Email",
                table: "Students",
                type: "nvarchar(max)",
                nullable: false,
                defaultValue: "");
        }

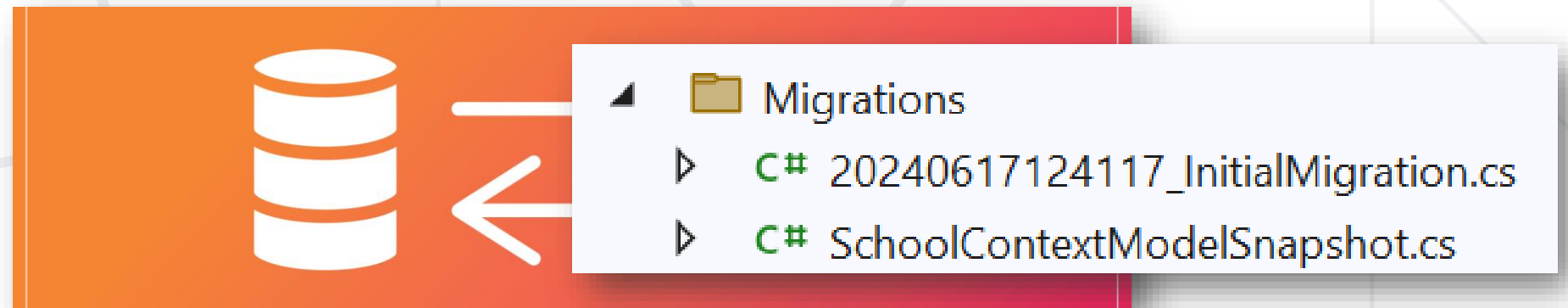
        0 references
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "Email",
                table: "Students");
        }
    }
}
```

Remove a Migration Before Applying It

- How to **remove a migration before applying it to the database**
 - Open the **Package Manager Console**
 - Run the '**remove-migration**' command
- Why Remove a Migration?
 - Realized **there's an error in the migration**
 - Need to **make additional changes** to the model
 - Created a migration **accidentally**



- The command **removes the last migration** that **has not been applied** to the database
- It **deletes the migration file** from the '**Migrations**' folder
- **No Database Changes**
 - Since the **migration has not been applied yet**, there are **no changes made** to the database



- After creating the '**AddStudentEmail**' migration, you realize you forgot to set the '**Email**' property as required
- Instead of applying the migration, you remove it, update the model, and create a new migration

```
public class Student
{
    0 references
    public int Id { get; set; }

    0 references
    public string Name { get; set; } = null!;

    0 references
    public int Age { get; set; }

    [Required]
    0 references
    public string Email { get; set; } = null!;
}
```



```
Package Manager Console
Package source: All
PM> remove-migration
Build started...
Build succeeded.
Removing migration '20240617124221_AddStudentEmail'.
Reverting the model snapshot.
Done.
PM> add-migration AddStudentEmail
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> |
```

Applying the Migration

- **Updating the database**
 - Run the '**update-database**' command
- **Verifying the changes**
 - Using **SQL Server Management Studio** (SSMS)
 - Open SSMS and **connect to the SQL Server instance**
 - Navigate to the '**School**' database and **refresh the tables**
 - Verify that the '**Students**' table now **includes the updated 'Email' column**



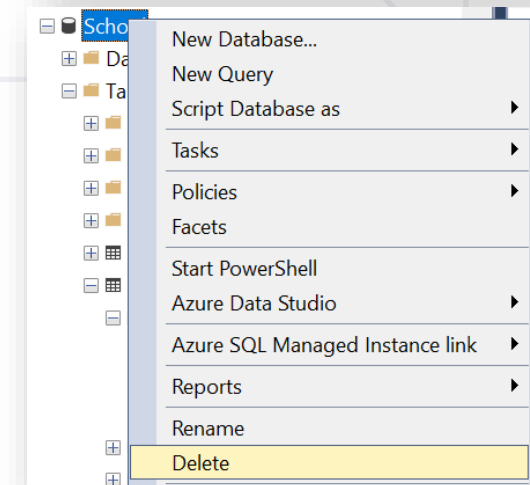
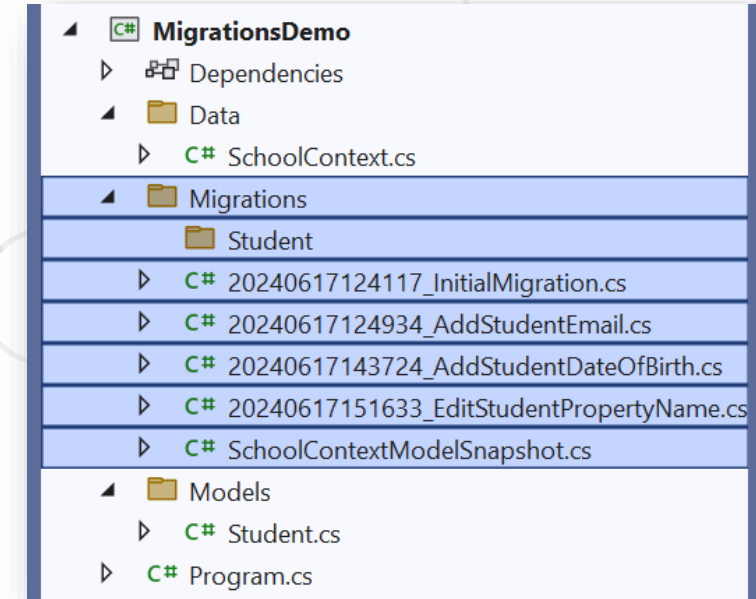
- All existing Migrations could be listed by executing '**Get-Migration**' command

```
PM> Get-Migration  
Build started...  
Build succeeded.
```

id	name	safeName	applied
--	----	-----	-----
20240617124117_InitialMigration	InitialMigration	InitialMigration	True
20240617124934_AddStudentEmail	AddStudentEmail	AddStudentEmail	False
20240617143724_AddStudentDateOfBirth	AddStudentDateOfBirth	AddStudentDateOfBirth	False
20240617151633_EditStudentPropertyName	EditStudentPropertyName	EditStudentPropertyName	False

Resetting All Migrations

- In some **extreme cases**, it may be necessary to **remove all migrations** and start over
- This can be easily done by **deleting** your '**Migrations**' folder and **dropping your database**
- At that point you **can create a new initial migration**, which will contain your entire current schema





Customize Migration Code

Review Migration Files

Why Customize Migrations

■ Real-World Requirements

- Default migration code may **not always meet specific** business or technical **requirements**
- Customizing migrations allows for **more precise control** over the database schema changes

■ Complex Operations

- Perform **complex data transformations** or ensure data integrity during schema changes



How to Customize Migrations

- After creating a migration, EF Core generates code in the **Up** and **Down** methods
- You can **modify this code** to suit your specific needs
- One example where customizing migrations is required is when **renaming a property**

```
public class Student
{
    0 references
    public int Id { get; set; }

    0 references
    public string FullName { get; set; } = null!;

    0 references
    public int Age

    [Required]
    0 references
    public string

    0 references
    public DateTime DateOfBirth { get; set; }
}
```

FullName

Rename will update 5 references in 5 files.

☐ Include comments

☒ Include strings

Enter to rename, Shift+Enter to preview

Create Custom Migration

- EF Core is **unable to know when the intention is to drop a column and create a new one**, and when a column should be renamed
- If the above migration is applied as-is, all your student **names will be lost**
- To rename a column, **replace the above generated migration** with the following

```
migrationBuilder.DropColumn(  
    name: "Name",  
    table: "Students");  
  
migrationBuilder.AddColumn<string>(  
    name: "Name",  
    table: "Students",  
    nullable: false,  
    defaultValue: "");
```

```
migrationBuilder.RenameColumn(  
    name: "Name",  
    table: "Students",  
    newName: "FullName");
```


Tip

- The **migration scaffolding process warns** when an operation might result in **data loss** (like dropping a column)
- If you see that warning, be especially sure to **review the migrations** code for accuracy



What Else Could Be Customized?

- Adding / Altering Indexes
- Custom SQL commands
- Adding / Altering Constraints
 - Adding a Unique Constraint
 - Adding a Foreign Key Constraint
- Data Seeding
- Advanced Column Types and Computed Columns





Multiple Providers

Does Db Matter?

Multiple Db Providers

- The **EF Core Tools** only scaffold migrations for the **active provider**
- It is possible to maintain **multiple sets of migrations**
 - One for each provider
 - Adding a **migration to each**, for **every model change**
- Using **multiple context** types
- Using **one context** type




Reasons to Work With Multiple Providers

- Maybe you are **building an application** distributed as a **Software-as-a-Service (SaaS)** product
 - And a standalone **product your customers** can install in their data center
- **SaaS** may use **PostgreSQL**
- Customers may have standardized on **Microsoft SQL Server**
- Some customers prefer to use **Windows Server** and **SQL Server** database and others prefer to use **Linux** and **MySQL**

- Add the **PostgreSQL provider** to your project



Npgsql.EntityFrameworkCore.PostgreSQL  by Shay R 6.0.29
PostgreSQL/Npgsql provider for Entity Framework Core. 8.0.4

- Updating the **DbContext**

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    var provider = Environment.GetEnvironmentVariable("PROVIDER");

    if (provider == "PostgreSQL")
    {
        optionsBuilder.UseNpgsql("Host=localhost;Database=School;Username=postgres;Password=postgres");
    }
    else
    {
        optionsBuilder.UseSqlServer("Server=localhost;Database=School;Trusted_Connection=True;");
    }
}
```

- Create subfolders
 - In your project, create a '**Migrations**' folder if it doesn't already exist
 - Within the '**Migrations**' folder, create subfolders for each provider, for example:
 - '**SqlServerMigrations**'
 - '**PostgreSqlMigrations**'

Setting Up Environment Variables

- **Environment variables** store configuration settings outside the codebase
- Setting Environment Variables:
 - For **SQL Server**
 - `$env:PROVIDER="SqlServer"`
 - For **PostgreSQL**
 - `$env:PROVIDER="PostgreSQL"`

Setting Up Environment Variables

- **Environment Variable** can be easily checked through the **Package Manager Console**:

- `echo $env:PROVIDER`

```
PM> $env:PROVIDER="PostgreSQL"
PM> echo $env:PROVIDER
PostgreSQL
PM>
```

- The Environment Variable value can be **switched** through the PM Console:

```
PM> $env:PROVIDER="PostgreSQL"
PM> echo $env:PROVIDER
PostgreSQL
PM> $env:PROVIDER="SqlServer"
PM> echo $env:PROVIDER
SqlServer
PM>
```

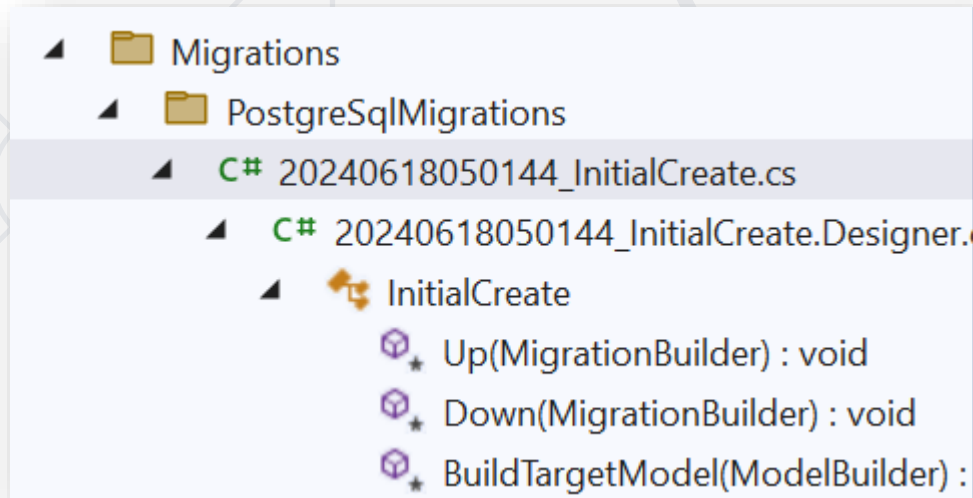
Creating a Migration for PostgreSQL

- Set the **correct Environment Variable value**
 - `$env:PROVIDER="PostgreSQL"`
- Create the migration and specify the output directory:
 - `add-migration InitialCreate -o Migrations/PostgreSqlMigrations`

```
PM> add-migration InitialCreate -o Migrations/PostgreSqlMigrations
Build started...
Build succeeded.
```

PostgreSQL Migration File

- The migration files are located in the specified subfolder



```
namespace MigrationsDemo.Migrations.PostgreSqlMigrations
{
    1 reference
    public partial class InitialCreate : Migration
    {
        0 references
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AlterColumn<string>(
                name: "FullName",
                table: "Students",
                type: "text",
                nullable: false,
                oldClrType: typeof(string),
                oldType: "nvarchar(max)");

            migrationBuilder.AlterColumn<string>(
                name: "Email",
                table: "Students",
                type: "text",
                nullable: false,
                oldClrType: typeof(string),
                oldType: "nvarchar(max)");

            migrationBuilder.AlterColumn<DateTime>(
                name: "DateOfBirth",
                table: "Students",
                type: "timestamp with time zone",
                nullable: false,
                oldClrType: typeof(DateTime),
                oldType: "datetime2");
        }
    }
}
```



Error Handling

Common Errors and Good Practices

Model and Database Schema Mismatch

- Occurs when the **model and the database are out of sync**
 - `System.InvalidOperationException`: The model backing the 'DbContext' context has changed since the database was created.
- Ensure all migrations are applied using '**Update-Database**'



- Conflicts between migrations **created by different team members**
- Migration **conflicts can also happen locally**
 - This can occur if you make **multiple changes** to your model and generate **migrations without applying them in sequence**
 - `System.Data.SqlClient.SqlException`: There is already an object named 'TableName' in the database.

- Solutions:
 - **Apply migrations in sequence** to keep the local database schema in sync with the model
 - Communicate and **coordinate migration creation** within the team
 - **Merge conflicting migrations manually** if necessary

- **Errors in the SQL commands** generated by EF Core
 - `System.Data.SqlClient.SqlException: Incorrect syntax near 'keyword'.`
- Solution:
 - Review and **customize the migration code to correct SQL syntax**
 - Use `'migrationBuilder.Sql'` to execute raw SQL if needed

- Risk of **data loss when dropping or renaming** columns incorrectly
- No specific error, but **data may be lost**
- Solution:
 - Use '**RenameColumn**' instead of dropping and adding columns
 - **Backup data** before applying risky migrations

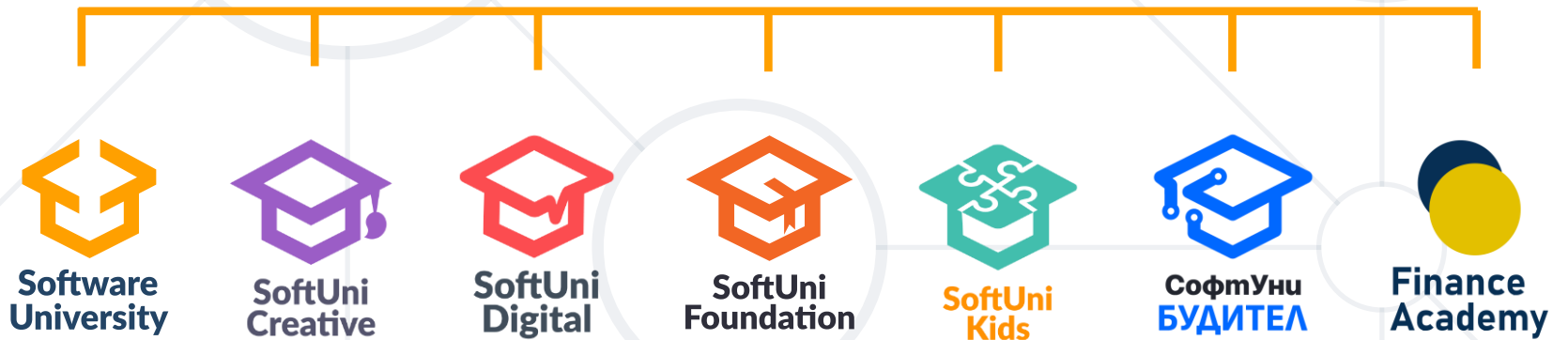
- Purpose and benefits of **migrations in EF Core**
- Essential **commands**
- Creating and applying migrations
- Customizing **migration code**
- Configuring and managing **connection strings**
- Organizing migrations and applying them in **sequence**
- Error Handling



Questions?



SoftUni



SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg, about.softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

