EF Core
Code First

Introduction

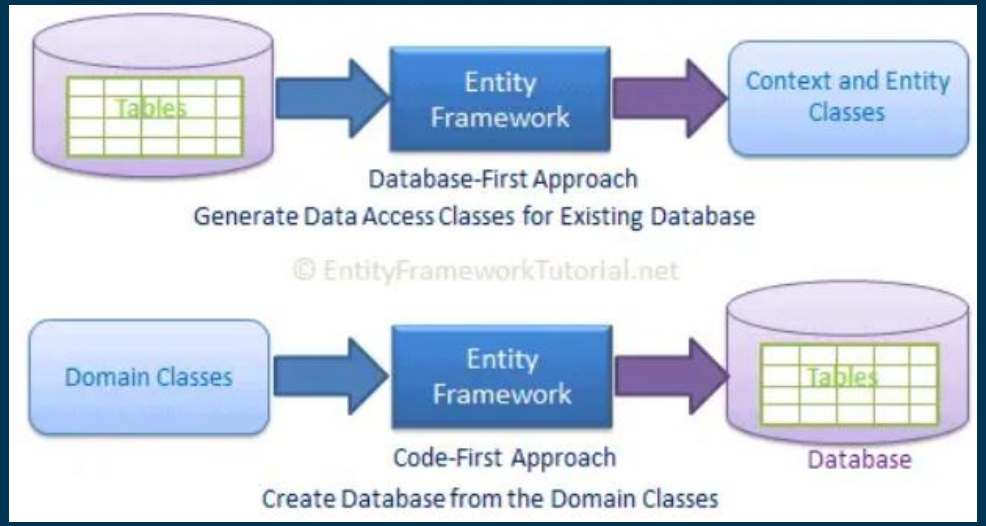
- Entity Framework Core is new version of Entity Framework after EF 6.x.
- It is open-source, lightweight, and a cross-platform version of Entity Framework data access technology.

- EF Core is intended to be used with .NET Core applications.
  - However, it can also be used with standard .NET 4.5+ framework based applications.

- Learn more on EF Core and EF 6 differences at:
  - https://learn.microsoft.com/en-us/ef/efcore-and-ef6/

EF Core Development Approaches

- EF Core supports two development approaches:
  - Code-First
  - Database-First



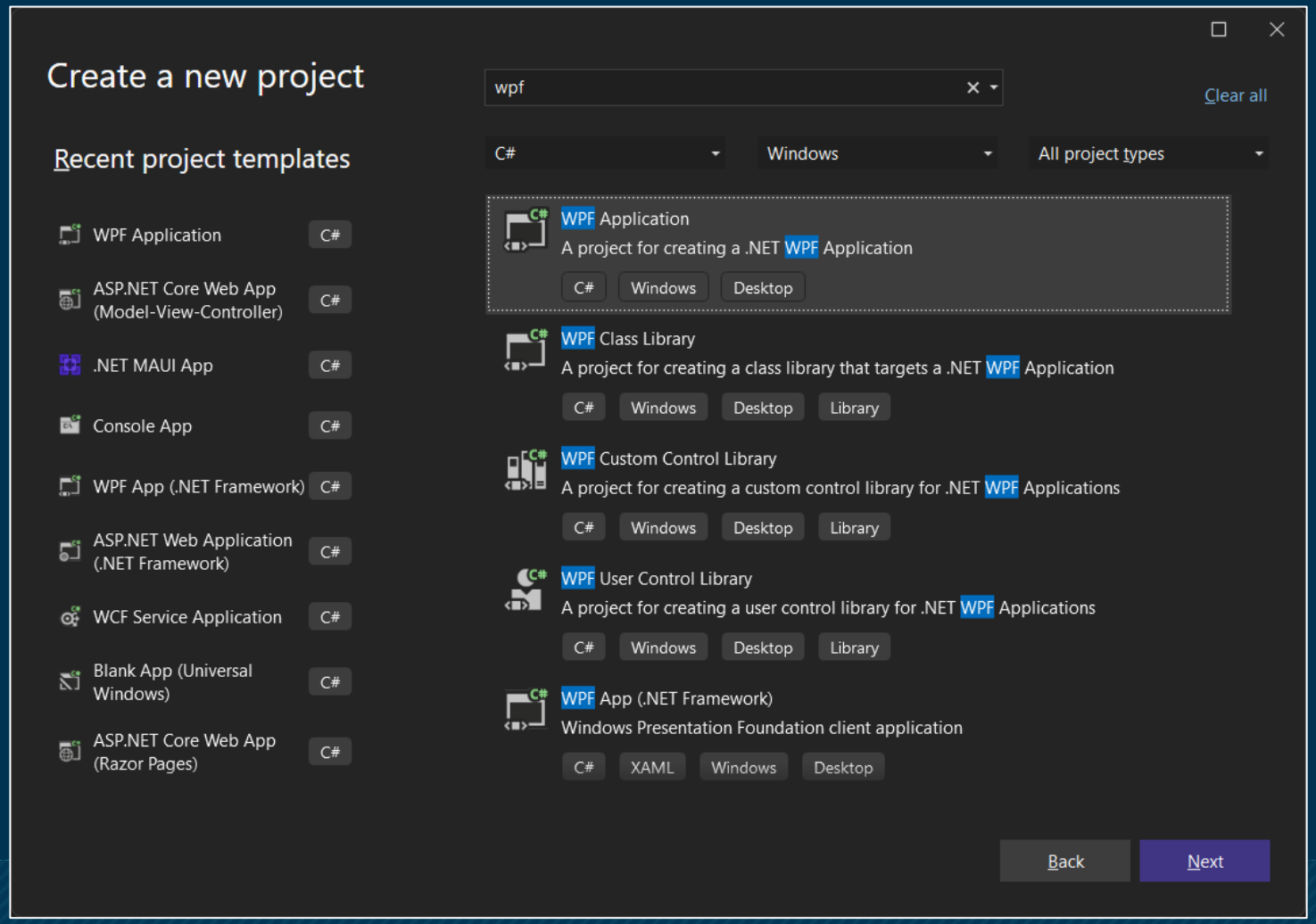
- EF Core mainly targets the code-first approach and provides little support for the database-first approach because the visual designer or wizard for DB model is not supported as of EF Core.

- In the code-first approach, EF Core API creates the database and tables using migration based on the conventions and configuration provided in your domain classes.
- This approach is useful in Domain Driven Design (DDD).

- In the database-first approach, EF Core API creates the domain and context classes based on your existing database using EF Core commands.
- This has limited support in EF Core as it does not support visual designer or wizard.

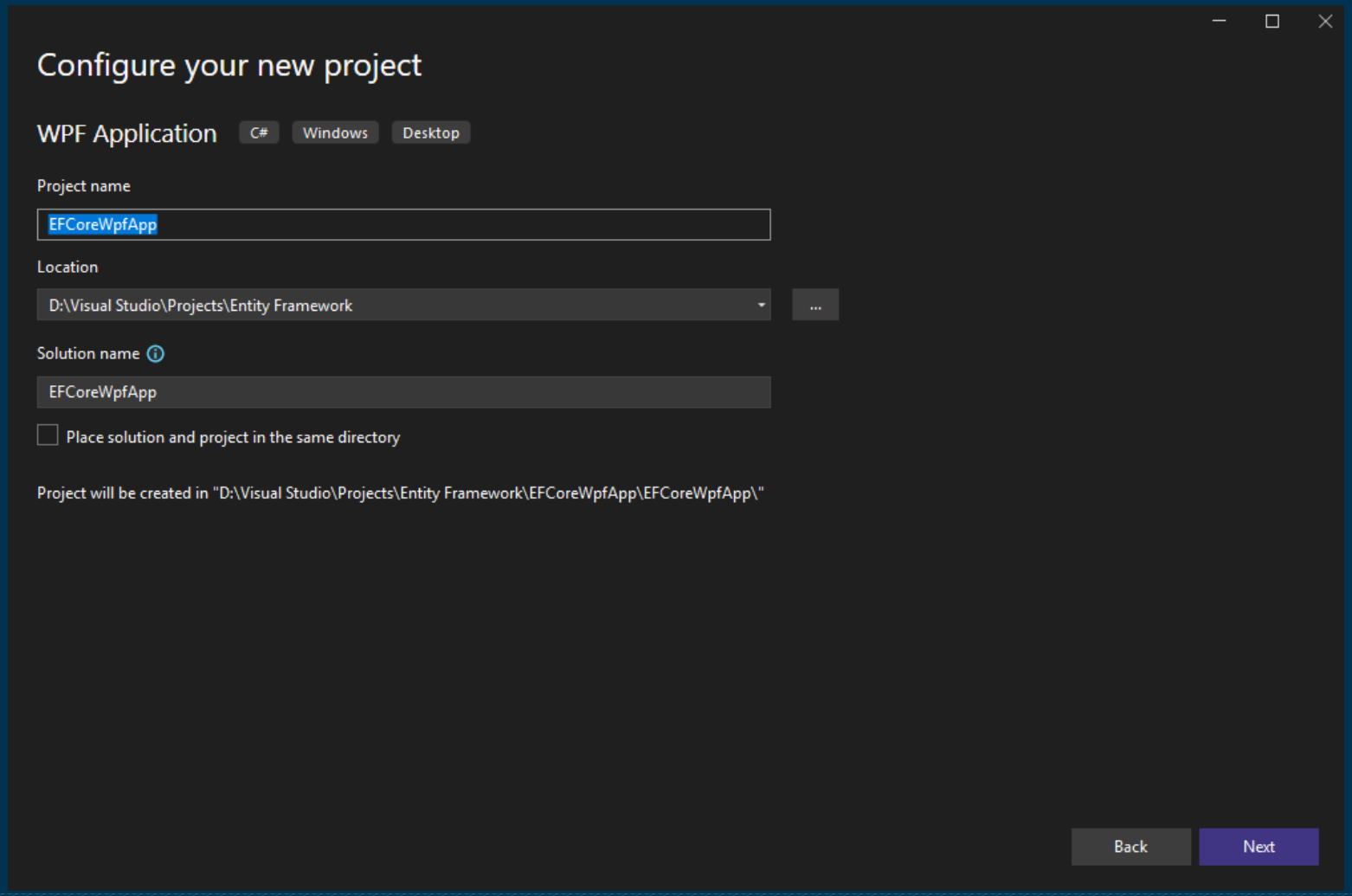
Create a WPF App

- Create a new WPF Application.
  - Make sure its **NOT** WPF App (.NET Framework).



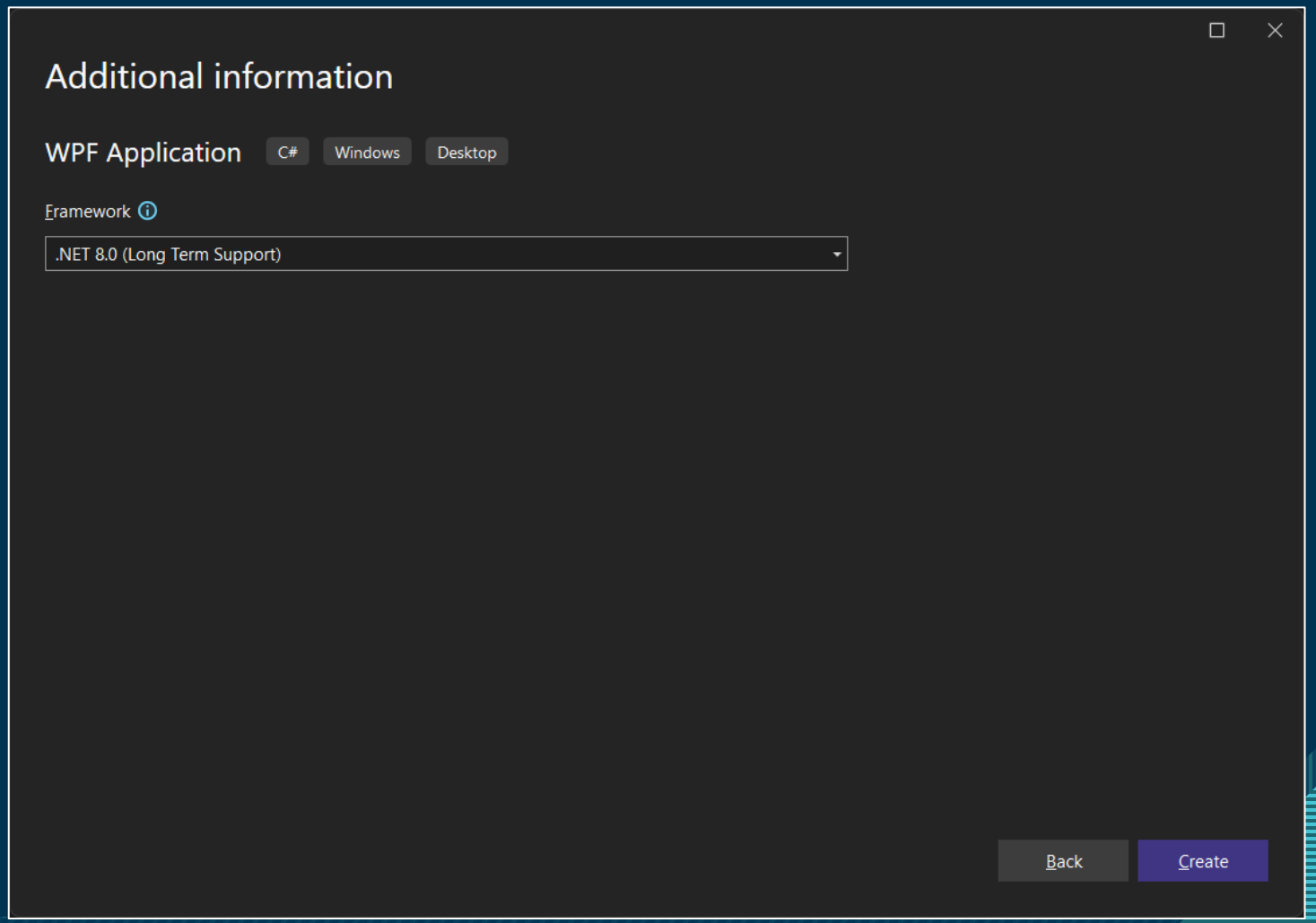
Create a WPF App

- Give it a name and location.



Create a WPF App

- Make a mental note of the .NET version.
- Later, when installing EF Core, you need to match EF Core's version with .NET version.



Install Entity Framework Core

- The `Microsoft.EntityFrameworkCore` is the base package for all the basic operations of EF Core.
- However, you have to install a database provider package from NuGet for the database you use in your project.
- For example, to use the MS SQL Server database, you need to install `Microsoft.EntityframeworkCore.SqlServer` package.

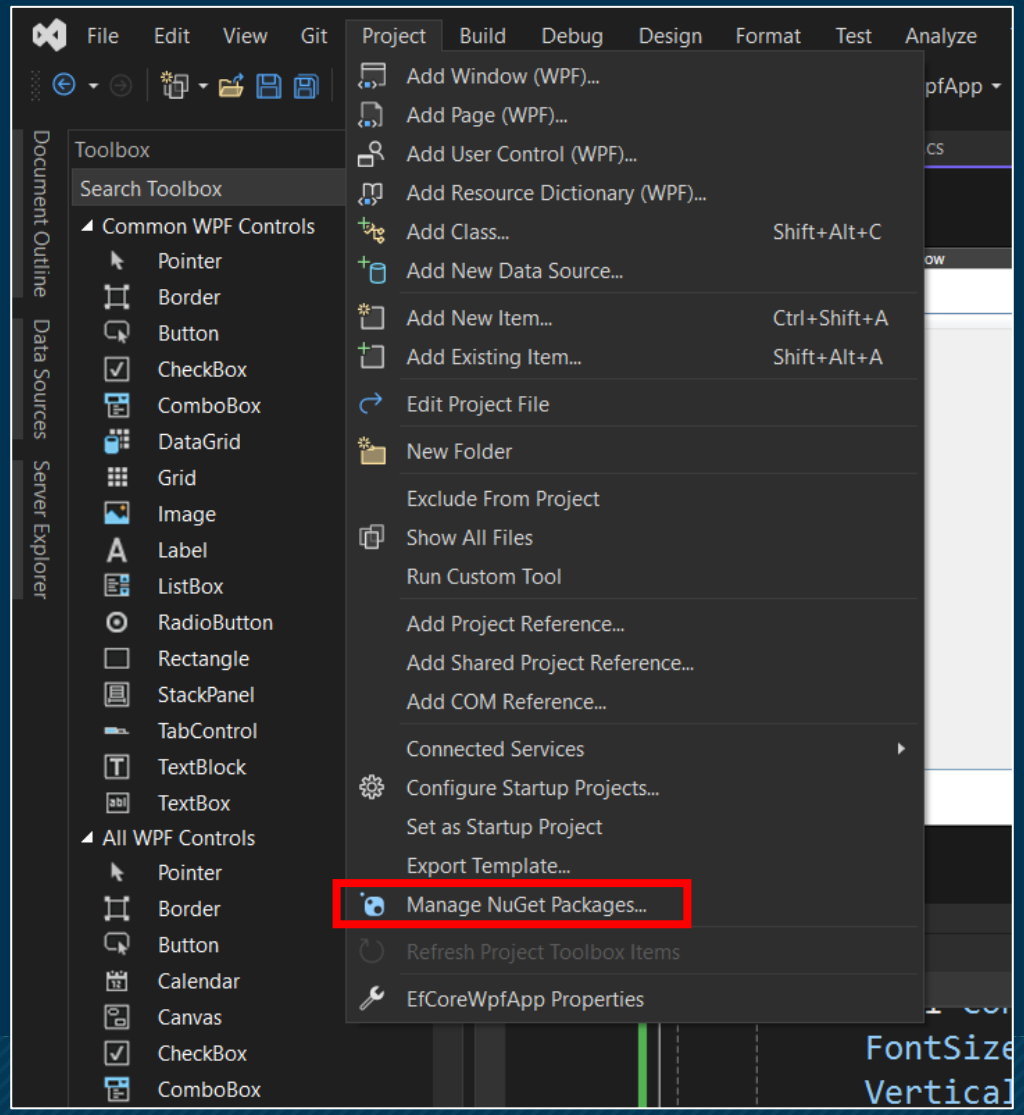
- The database provider package `Microsoft.EntityframeworkCore.SqlServer` includes all the other dependent packages it needs.
  - So, it includes `Microsoft.EntityFrameworkCore` too.
  - There is no need to install it separately.

For MySQL

- For MySQL, install the following NuGet packages.
  - Pomelo.EntityFrameworkCore.MySql: A popular provider for MySQL
  - Microsoft.EntityFrameworkCore.Design: For EF Core tools
  - Microsoft.EntityFrameworkCore.Tools: For EF Core migration commands

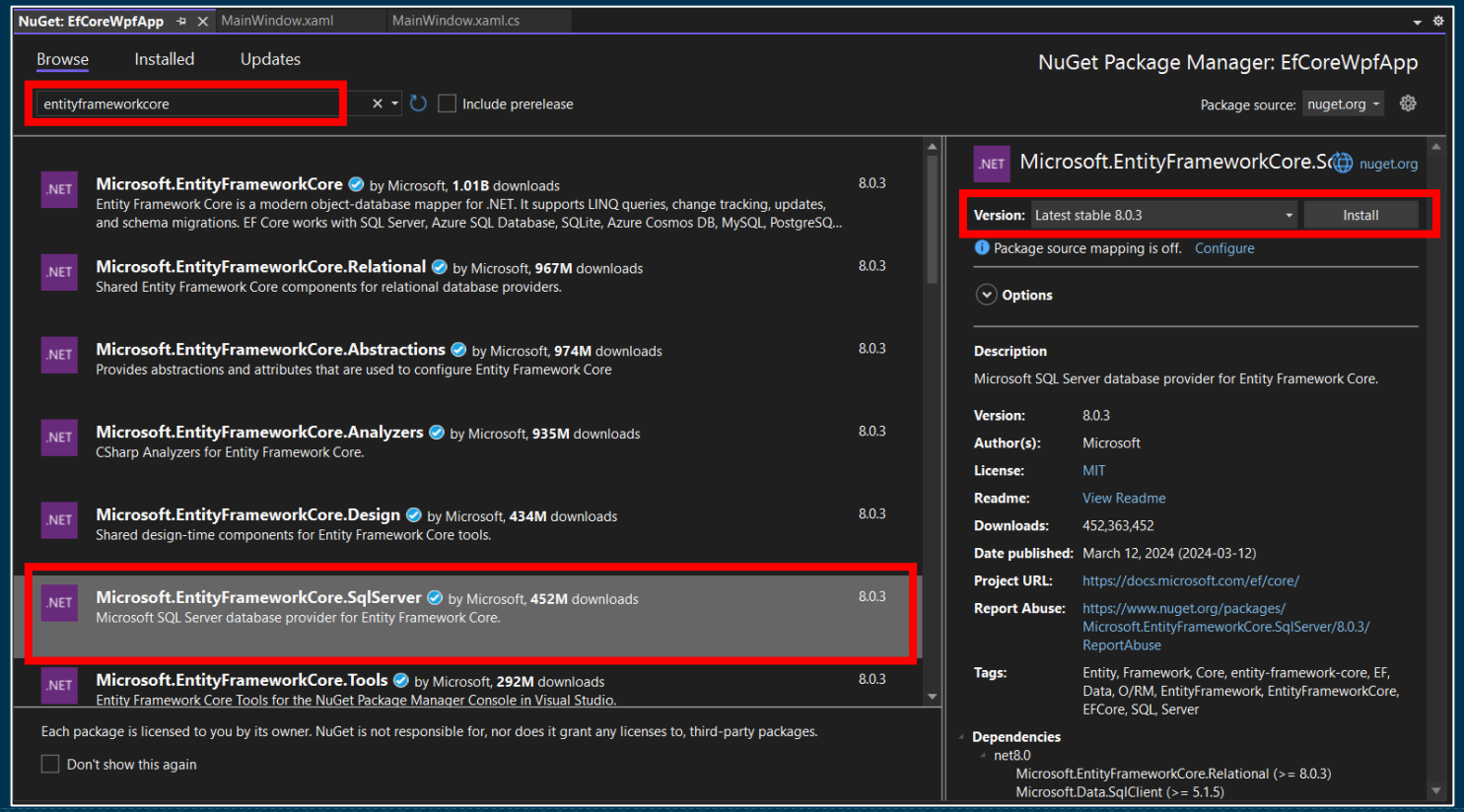
Install Microsoft.EntityframeworkCore.SqlServer

- Go to Project → Manage NuGet Packages….



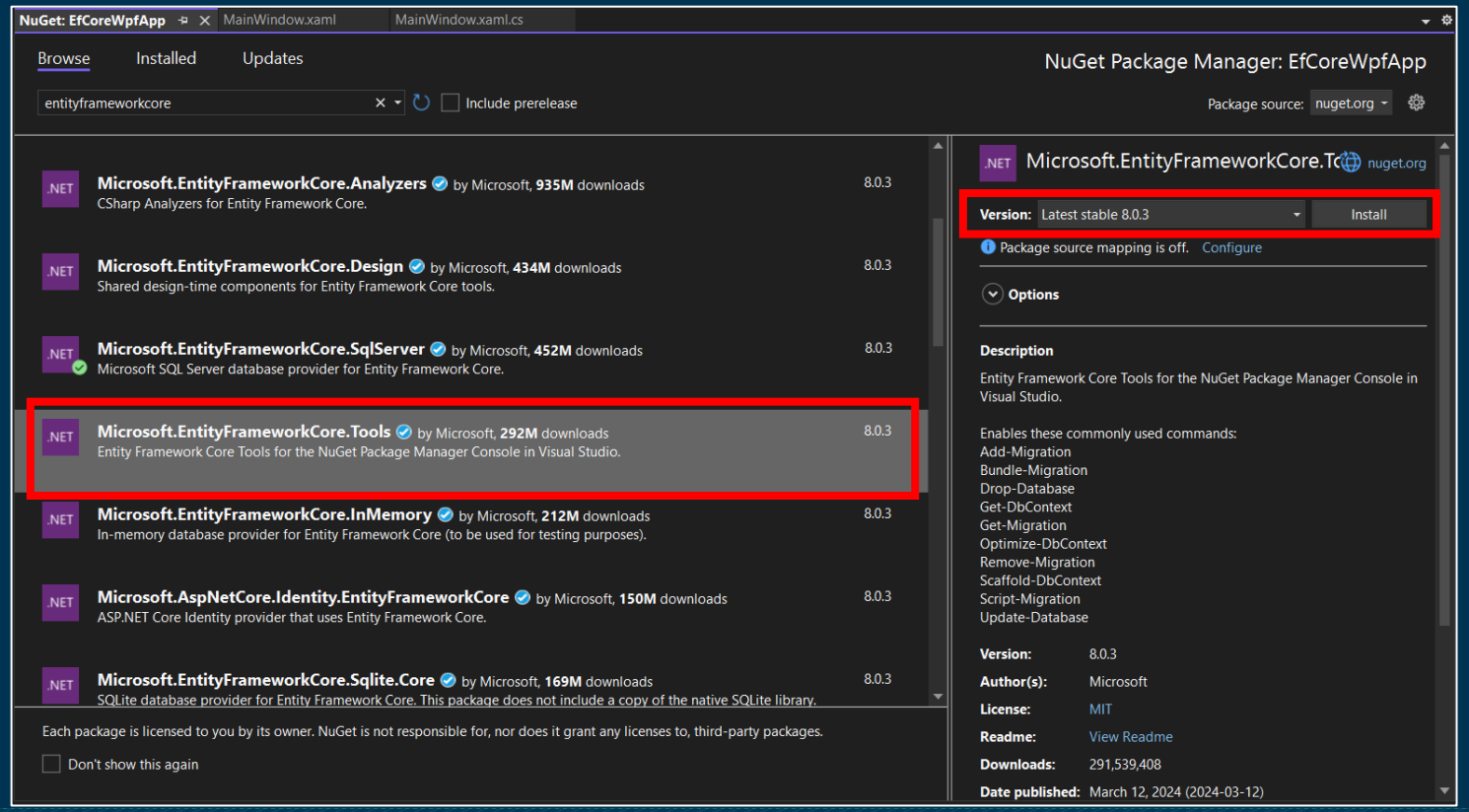
Install Microsoft.EntityframeworkCore.SqlServer

- Search for entityframeworkcore.
- Select and install `Microsoft.EntityframeworkCore.SqlServer` package.
- Before installing, match the package version with the .NET version.



Install Microsoft.EntityframeworkCore.Tools

- Also, install `Microsoft.EntityframeworkCore.Tools` package.
- This package is required for Migrations – discussed later.
- Before installing, match the package version with the .NET version.



Create Entities in Entity Framework Core

- We are building a school application so here we will create two classes `Student` and `Standard` where every `Student` is associated with one `Standard`.
- And the `Standard` can have a collection of `Students`.
- This is a *one-to-many relationship*.

```
public class Student
{
    // scalar properties
    public int StudentId { get; set; }
    public string? StudentName { get; set; }
    public int? StandardId { get; set; }

    // navigation property
    public Standard Standard { get; set; }
}
```

```
public class Standard
{
    // scalar properties
    public int StandardId { get; set; }
    public string? StandardName { get; set; }

    // navigation property
    public ICollection<Student> Students { get; set; }
}
```

DbContext in Entity Framework Core

- The `DbContext` class is an integral part of the Entity Framework.
- An instance of `DbContext` represents a session with the database which can be used to query and save instances of your entities to a database.
- To use `DbContext` in our application, create a class that derives from `DbContext`.

```
using Microsoft.EntityFrameworkCore;

public class SchoolContext : DbContext
{

}
```

DbContext in Entity Framework Core

- Add the domain classes as `DbSet<TEntity>` type properties.

```
using Microsoft.EntityFrameworkCore;

public class SchoolContext : DbContext
{
    // define entity sets
    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }
}
```

Configure Database Connection

- So far, we haven't specified the database name and database server info yet.
- We can override `DbContext`'s `OnConfiguring()` method and specify the connection string for the database.
- This method is called for each instance of the context that is created.

```
public class SchoolContext : DbContext
{
    // define entity sets
    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }

    // define the database connection
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(LocalDB)\MSSQLLocalDB;
                                      Database=SchoolEFCore;
                                      Trusted_Connection=True;
                                      MultipleActiveResultSets=True;");
    }
}
```

Configure Database Connection for MySQL

- If you're working with MySQL, ensure that the Pomelo.EntityFrameworkCore.MySql package is installed.
- Call the method `UseMySql` on the `optionsBuilder`.
- And define the connection string.

```
using Microsoft.EntityFrameworkCore;

public class SchoolContext : DbContext
{
    // define entity sets
    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }

    // define the database connection
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseMySql(@"server=localhost;port=3306;database=school;
                                  user id=root;password=root;");
    }
}
```

Data Seeding

- Data seeding is the process of populating a database with an initial set of data.
- In EF Core, seeding data can be associated with an entity type as part of the model configuration.
- Then EF Core migrations can automatically compute what insert, update or delete operations need to be applied when upgrading the database to a new version of the model.

- Learn more:
  - https://learn.microsoft.com/en-us/ef/core/modeling/data-seeding

Data Seeding

- As an example, this will configure seed data for a `Student` in `OnModelCreating`:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>().HasData(new Student { StudentId = 1, Name = "John" });
}
```

- To add entities that have a relationship, the foreign key values need to be specified:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>().HasData(new Student { StudentId = 1, Name = "John", StandardId = 1 });
}
```

- Once the data has been added to the model, migrations should be used to apply the changes.

Data Seeding

- Now, populate the database with an initial set of data.
- Add some sample `Standard`s and `Student`s by overriding the `OnModelCreating` method in the `StudentContext` class.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Standard>().HasData(
        new Standard { StandardId = 1, Name = "Standard 1" },
        new Standard { StandardId = 2, Name = "Standard 2" },
        new Standard { StandardId = 3, Name = "Standard 3" }
    );

    modelBuilder.Entity<Student>().HasData(
        new Student { StudentId = 1, Name = "John", StandardId = 1 },
        new Student { StudentId = 2, Name = "Anne", StandardId = 1 },
        new Student { StudentId = 3, Name = "Mark", StandardId = 2 },
        new Student { StudentId = 4, Name = "Tina", StandardId = 3 }
    );
}
```

Database Schema Creation in EF Core

- After creating the context and entity classes, it's time to start interacting with our underlying database using the context class.
- However, before we save or retrieve data from the database, we first need to ensure that the database schema is created as per our entities and configurations.

- You can create a database and schema by using Migrations.

- Migrations allows you to create an initial database schema based on your entities.
- And then as and when you add / delete / modify your entities and config, it will sync to the corresponding schema changes to your database so that it remains compatible with your EF Core model.

Migrations in Entity Framework Core

- Migration is a way to keep the database schema in sync with the EF Core model by preserving data.

- EF Core API builds the model from the entity classes and the `DbContext` class.
- EF Core Migrations API will create or update the database schema based on the EF Core model.
- Whenever you change the domain classes, you need to run migration commands to keep the database schema up to date.
- EF Core migrations are a set of commands which you can execute in Package Manager Console.

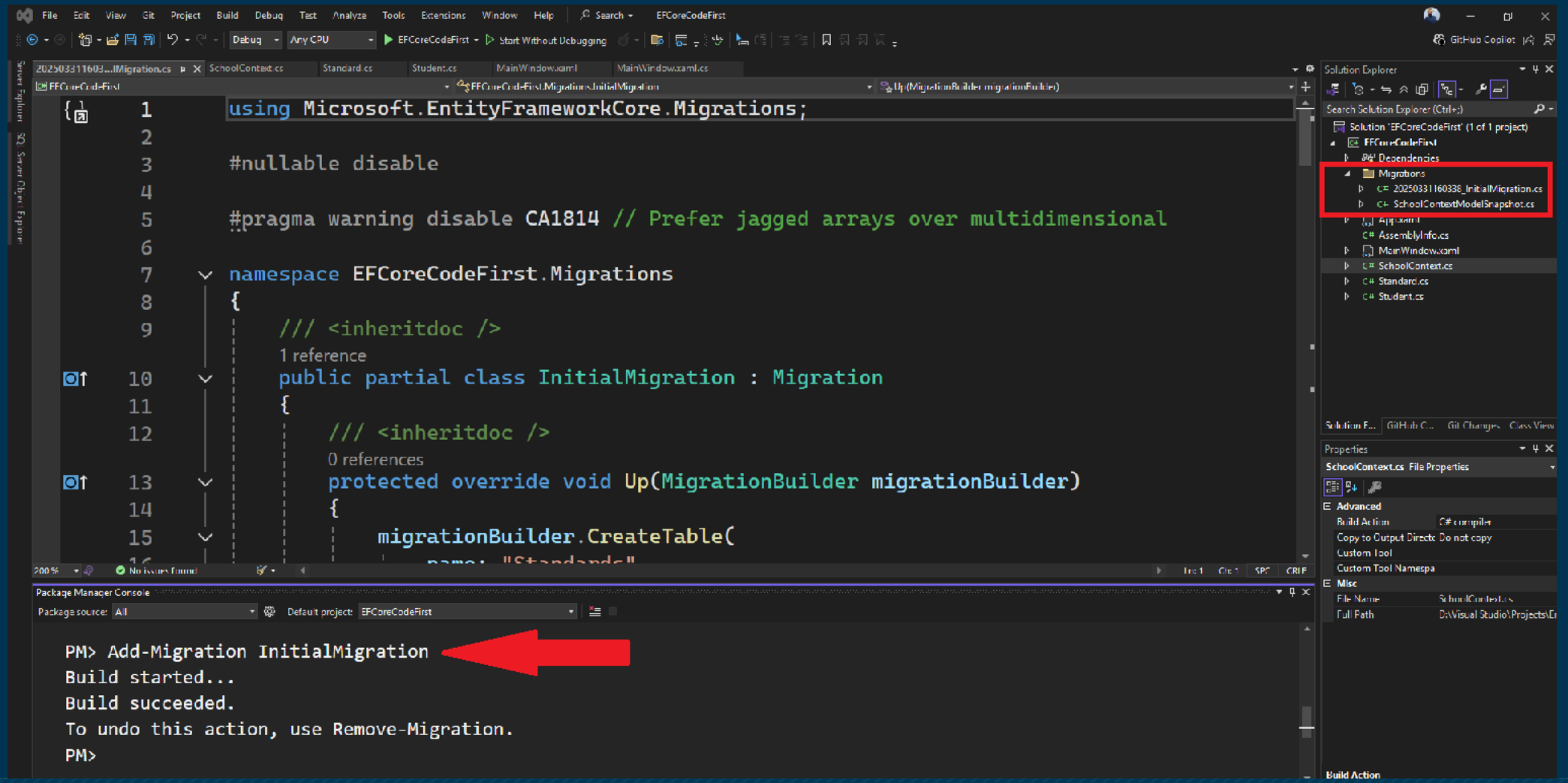
Adding a Migration

- EF Core provides migrations commands to create, update, or remove tables and other DB objects based on the entities and configurations.
- At this point, there is no SchoolEFCore database.
- So, we need to create the database from the model by adding a migration.
- In Visual Studio, open NuGet Package Manager Console from Tools → NuGet Package Manager → Package Manager Console and enter the following command:

```
PM> Add-Migration InitialMigration
```

Adding a Migration

- This will create a new folder named Migrations in the project and create the ModelSnapshot files.



Adding a Migration

- The `Add-Migration` command does not create the database.
- It just creates the two snapshot files in the Migrations folder.
  - `<timestamp>_<MigrationName>.cs`: The main migration file which includes migration operations in the `Up()` and `Down()` methods.
    - The `Up()` method includes the code for creating DB objects.
    - The `Down()` method includes code for removing DB objects.
  - `<contextclassname>ModelSnapshot.cs`: A snapshot of your current model.
    - This is used to determine what changed when creating the next migration.
- Now, to create a database, use the `Update-Database` command in the Package Manager Console.

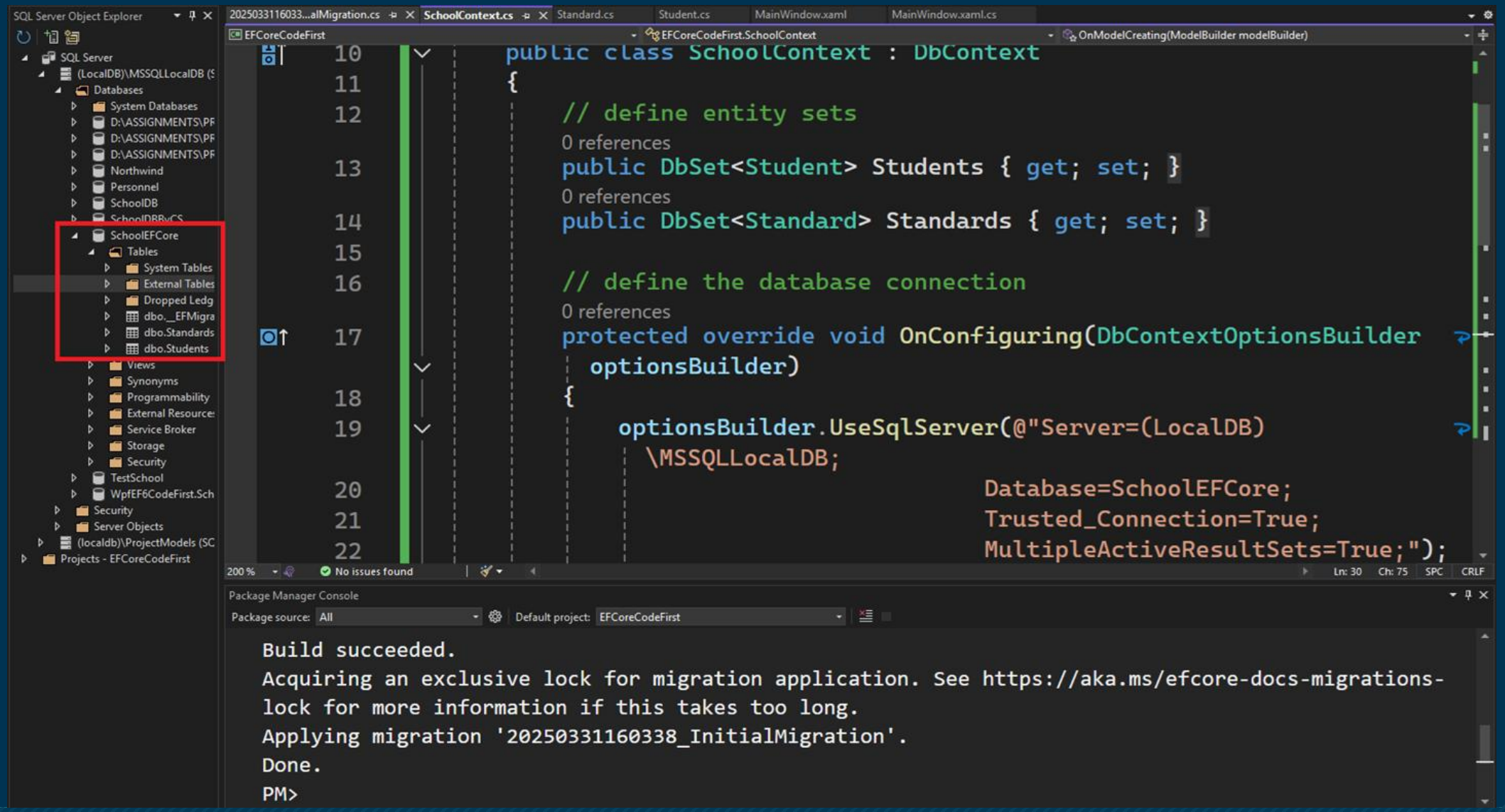
```
PM> Update-Database
```

Adding a Migration

- The `Update-Database` command creates the database.
- It creates a database with the name and location specified in the connection string in the `UseSqlServer()` method.
- It creates a table for each entity, `Student` and `Standard`.
- It also creates `_EFMigrationHistory` table that stores history of migrations applied overtime.

- See the screenshot on the next slide.

Adding a Migration



Adding a Migration

- Check out the data in the Students and Standards table.

dbo.Students [Data]

Max Rows: 1000

	StudentId	Name	StandardId
▷	1	John	1
	2	Anne	1
	3	Mark	2
	4	Tina	3
✱	*NULL*	*NULL*	*NULL*

dbo.Standards [Data]

Max Rows: 1000

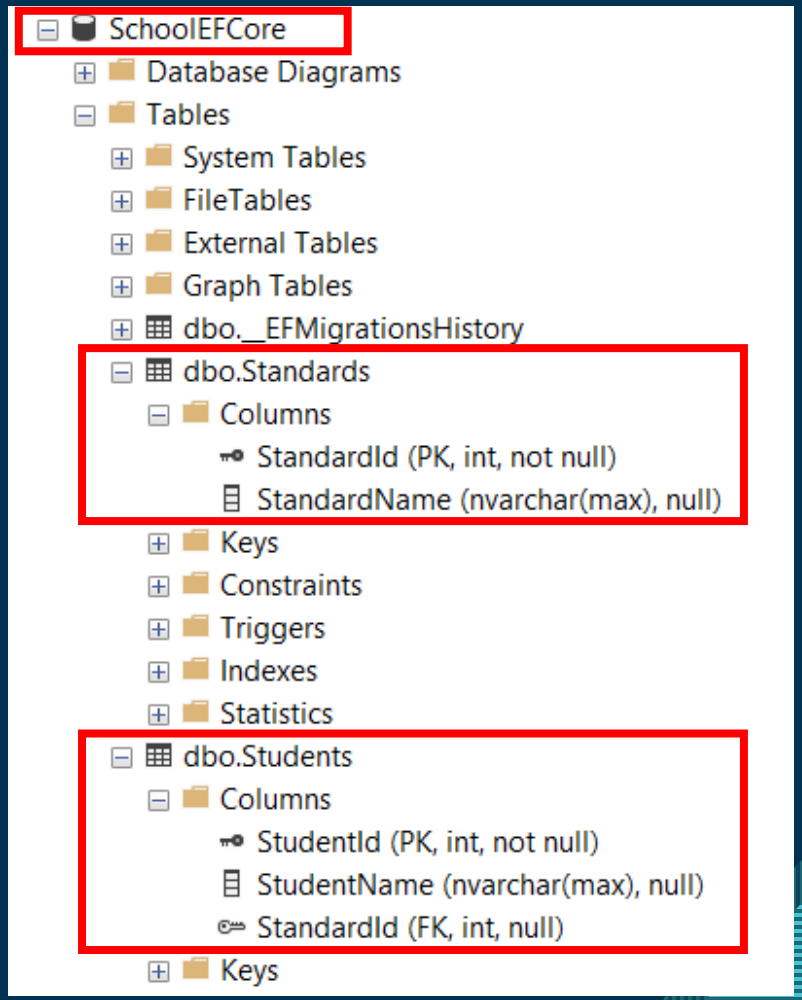
	StandardId	Name
▷	1	Standard 1
	2	Standard 2
	3	Standard 3
✱	*NULL*	*NULL*

Database Creation

- ***Where is the database and what are the tables and their columns?***

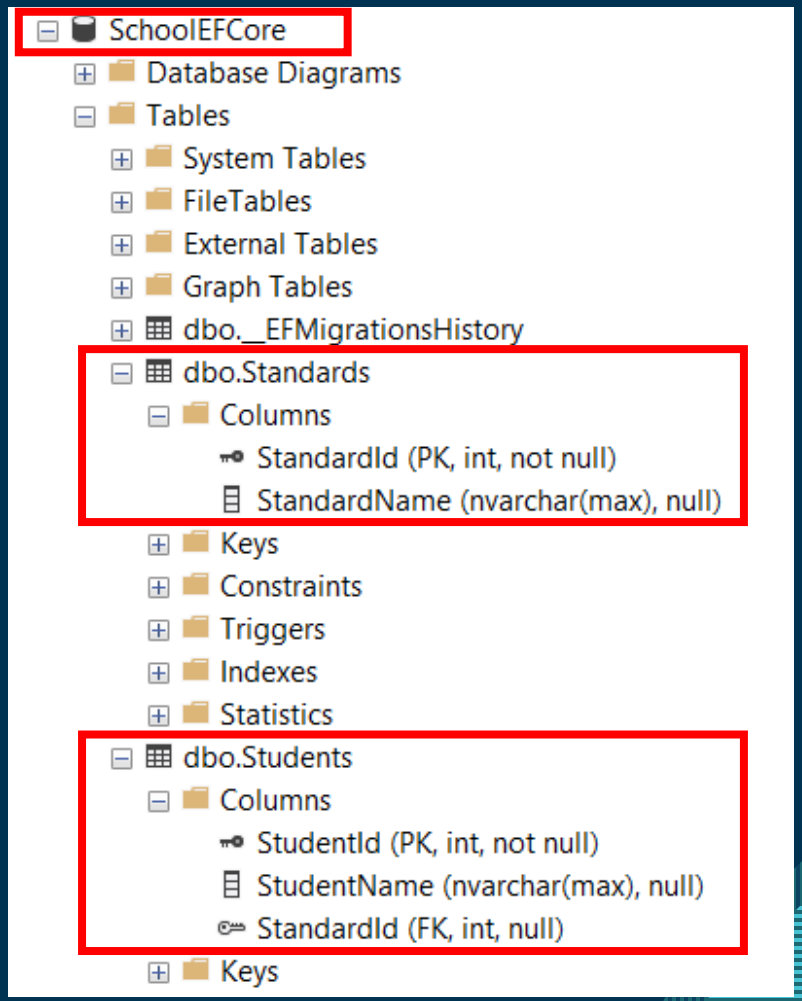
- This is the beauty of EF Code-First API.
- It creates the database based on the connection string in the context class.

- It also created two tables in this database, Students and Standards based on the `Student` and the `Standard` domain classes defined earlier.



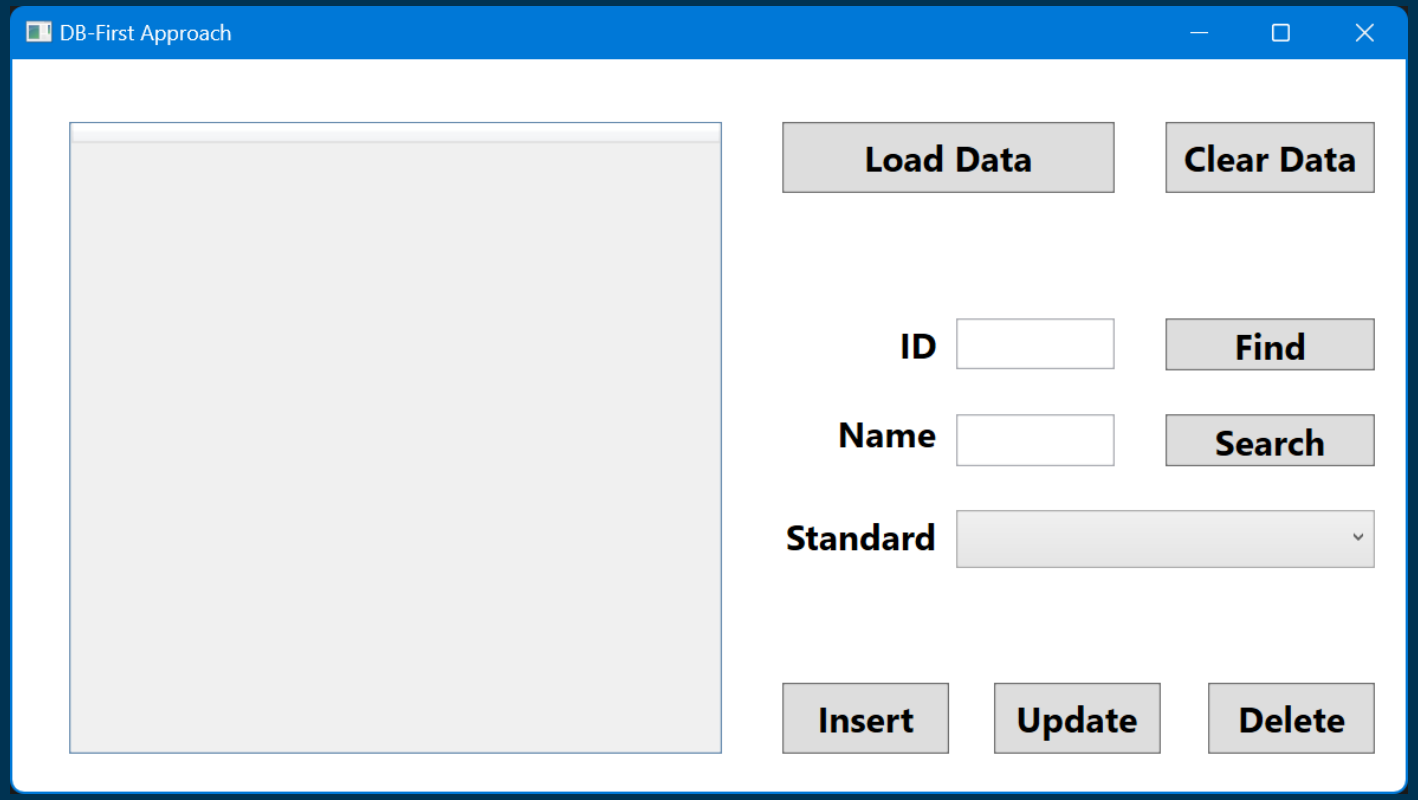
Database Creation

- As you can see here, it has created Students and Standards tables, and each table contains columns with appropriate datatype and length.
- The column names and datatype match the properties of the respective domain classes.
- It has also made StudentId and StandardId as PKs (primary keys) and created StandardId column as FK (foreign key).
- This way, without creating a database first, you can start writing an application that will eventually create a database from your domain classes.



Exercise

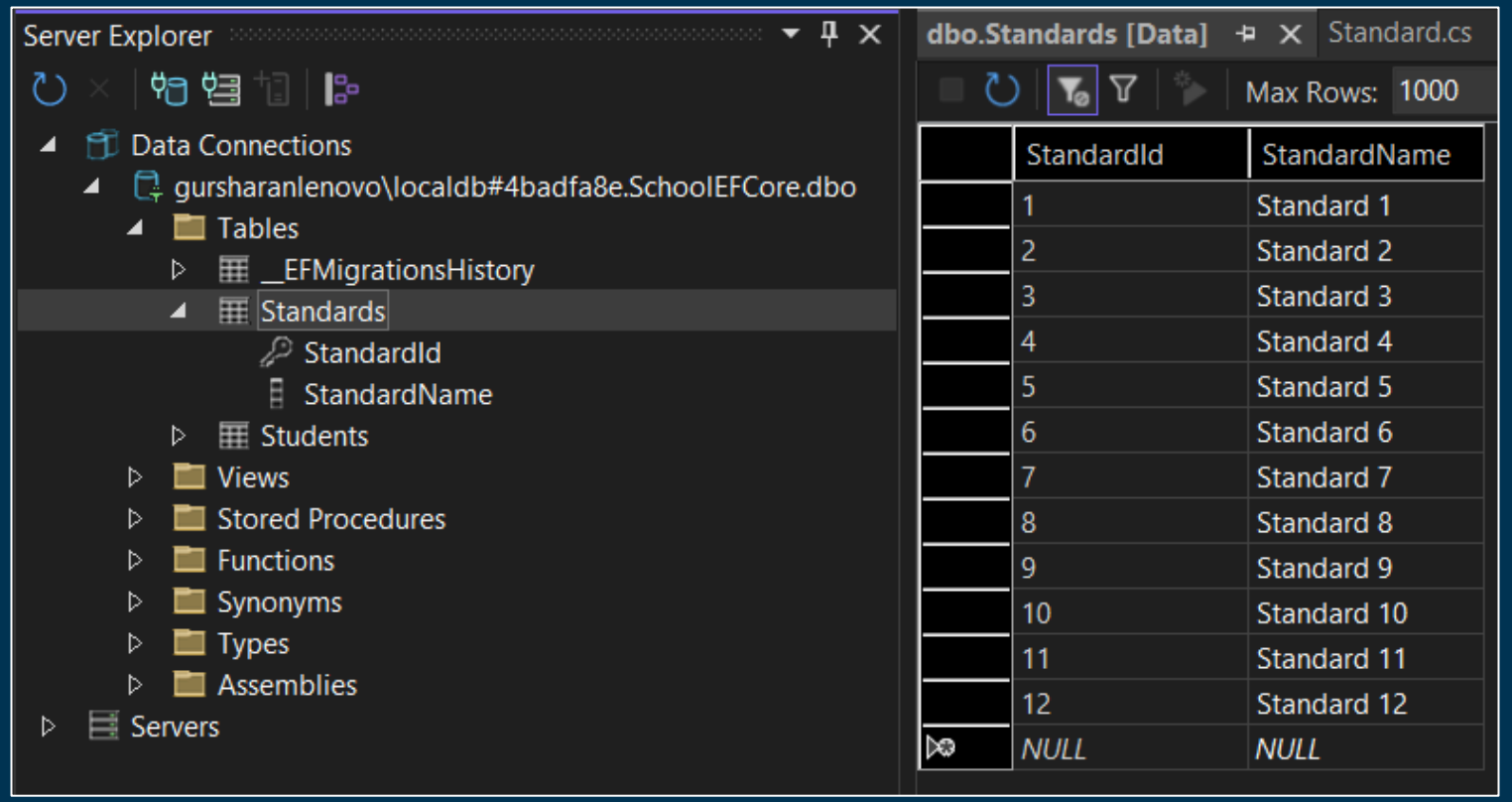
- Design the window something like this and implement its functionality.



- Copy-paste the XAML code from SLATE.

Exercise

- Add some data to the Standard table.



Exercise

- Implement the MainWindow's Loaded event.
- Populate the Standard ComboBox with standard names.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    using (var db = new SchoolContext())
    {
        var standards = db.Standards.ToList();

        cmbStandard.ItemsSource = standards;
        cmbStandard.DisplayMemberPath = "StandardName";
        cmbStandard.SelectedValuePath = "StandardId";
    }
}
```

Exercise

- Implement the Insert button's click event.

```
private void btnInsert_Click(object sender, RoutedEventArgs e)
{
    using (var db = new SchoolContext())
    {
        Student std = new Student();
        std.StudentName = txtName.Text;
        std.StandardId = (int)cmbStandard.SelectedValue;

        db.Students.Add(std);
        db.SaveChanges();

        MessageBox.Show("New student added");
    }
}
```

Exercise

- Run the app and insert a new student record.
- Verify that the record has been added to the database.

dbo.Students [Data]

Max Rows: 1000

StudentId	Name	StandardId
1	John	1
2	Anne	1
3	Mark	2
4	Tina	3
5	Bill	3
NULL	NULL	NULL

Apply Migrations for Modified Entities / Configurations

- Suppose we add some new entities or modify an existing entities or changed any configuration, then we again need to execute the `Add-Migration` and `Update-Database` commands to apply the changes to the database schema.
- For example, let's modify the `Student` entity and add `Age` property:

```
public class Student
{
    public int StudentId { get; set; }
    public string? StudentName { get; set; }
    public int? StandardId { get; set; }
    public int? Age { get; set; }

    public Standard? Standard { get; set; }
}
```

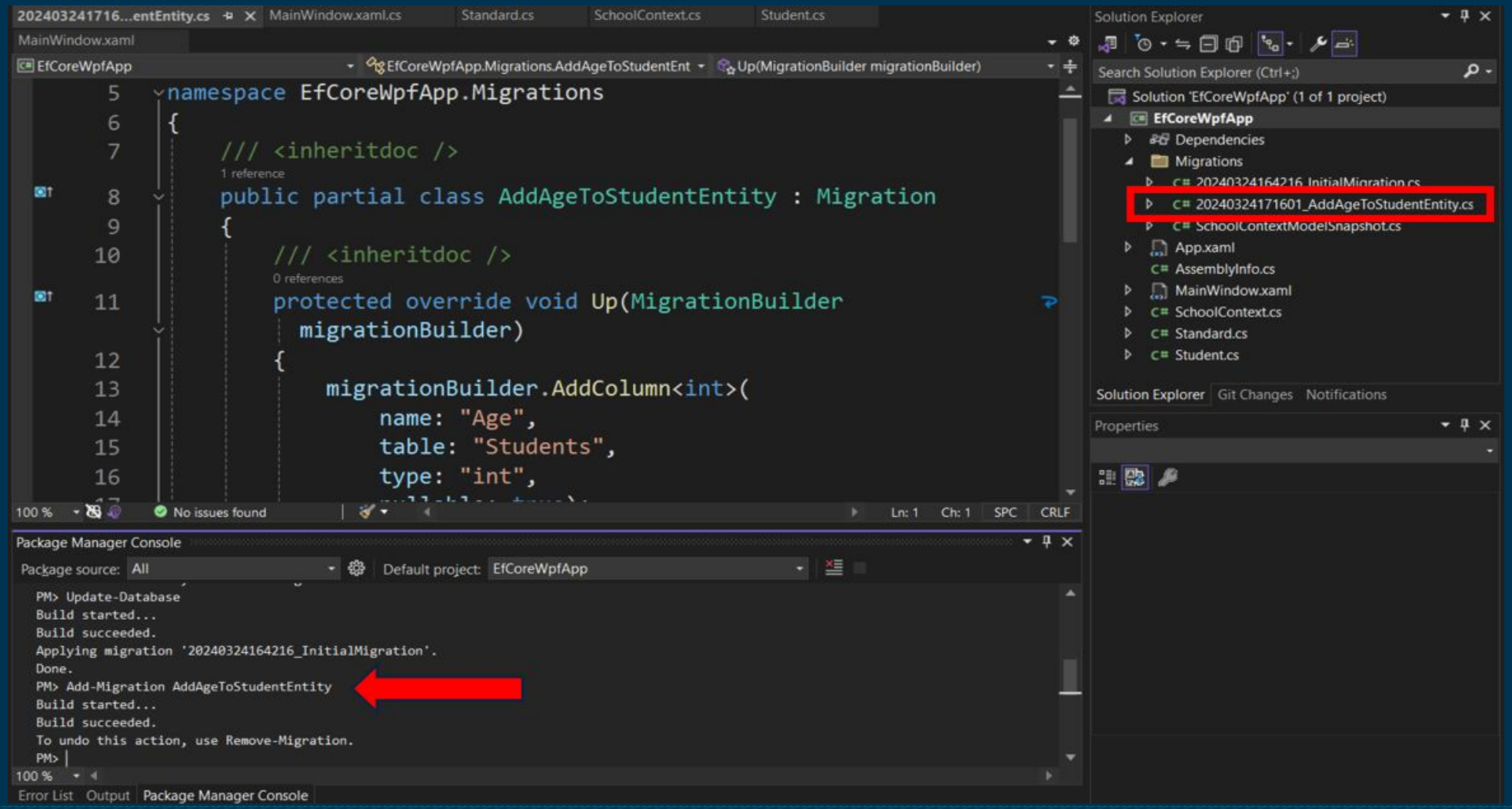
Apply Migrations for Modified Entities / Configurations

- Now, to sync our SchoolEFCore database with this change, execute the following commands:

```
PM> Add-Migration AddAgeToStudentEntity
```

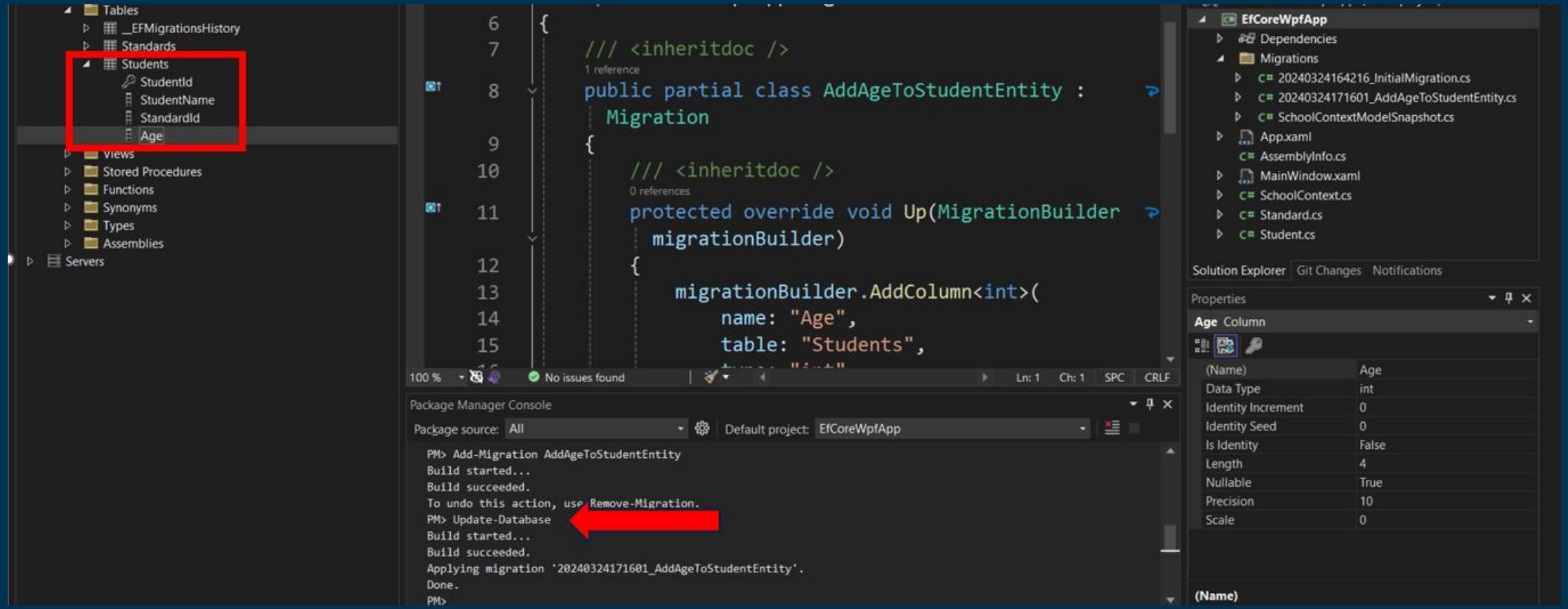
Apply Migrations for Modified Entities / Configurations

- This will generate another snapshot in the Migrations folder.



Apply Migrations for Modified Entities / Configurations

- Now, to update the database schema, execute the `Update-Database` command in PMC.
- This will add the column in the Student table.



EF Code-First Conventions

- Conventions are sets of default rules which automatically configure a conceptual model based on your domain classes when working with the Code-First approach.
- As you have seen in the previous example, EF configured PrimaryKeys, ForeignKeys, relationships, column data types etc. from the domain classes without any additional configurations.
- This is because EF Code-First leverages a programming pattern referred to as convention over configuration.
- If the conventions are followed in domain classes, then the database schema will be configured based on them.

EF Code-First Conventions

- The following table lists default code-first conventions:

Default Convention For	Description
Table Name	<Entity Class Name> + 's' EF will create a DB table with the entity class name suffixed by 's' e.g. `Student` domain class (entity) would map to the Students table.
Primary key Name	1) Id 2) <Entity Class Name> + "Id" (case insensitive) EF will create a primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive).
Foreign key property Name	By default EF will look for the foreign key property with the same name as the principal entity primary key name. If the foreign key property does not exist, then EF will create an FK column in the Db table with <Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name> e.g. EF will create Grade_GradeId foreign key column in the Students table if the Student entity does not contain foreignkey property for Grade.
Null column	EF creates a null column for all reference type properties and nullable primitive properties e.g. string, Nullable<int>, Student, Grade (all class type properties)
Not Null Column	EF creates NotNull columns for Primary Key properties and non-nullable value type properties e.g. int, float, decimal, datetime etc.
DB Columns order	EF will create DB columns in the same order like the properties in an entity class. However, primary key columns would be moved first.

Configure Domain Classes in EF Core

- Code-First builds the conceptual model from your domain classes using default conventions.
- EF Code-First leverages a programming pattern referred to as convention over configuration.
- However, you can override these conventions by configuring your domain classes to provide EF with the information it needs.
- This can be done using Data Annotation Attributes.

Data Annotations Attributes

- Data Annotations is a simple attribute-based configuration, which you can apply to your domain classes and its properties.
- This example demonstrates the use of some data annotation attributes:

```
[Table("StudentMaster")]
public class Student
{
    public Student() { }

    [Key]
    public int SID { get; set; }

    [Column("Name", TypeName="ntext")]
    [MaxLength(20)]
    public string StudentName { get; set; }

    [NotMapped]
    public int? Age { get; set; }

    public int StdId { get; set; }

    [ForeignKey("StdId")]
    public virtual Standard Standard { get; set; }
}
```

System.ComponentModel.DataAnnotations Attributes

Attribute	Description
`Key`	Can be applied to a property to specify a key property in an entity and make the corresponding column a PrimaryKey column in the database.
`Timestamp`	Can be applied to a property to specify the data type of a corresponding column in the database as rowversion.
`ConcurrencyCheck`	Can be applied to a property to specify that the corresponding column should be included in the optimistic concurrency check.
`Required`	Can be applied to a property to specify that the corresponding column is a NotNull column in the database.
`MinLength`	Can be applied to a property to specify the minimum string length allowed in the corresponding column in the database.
`MaxLength`	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.

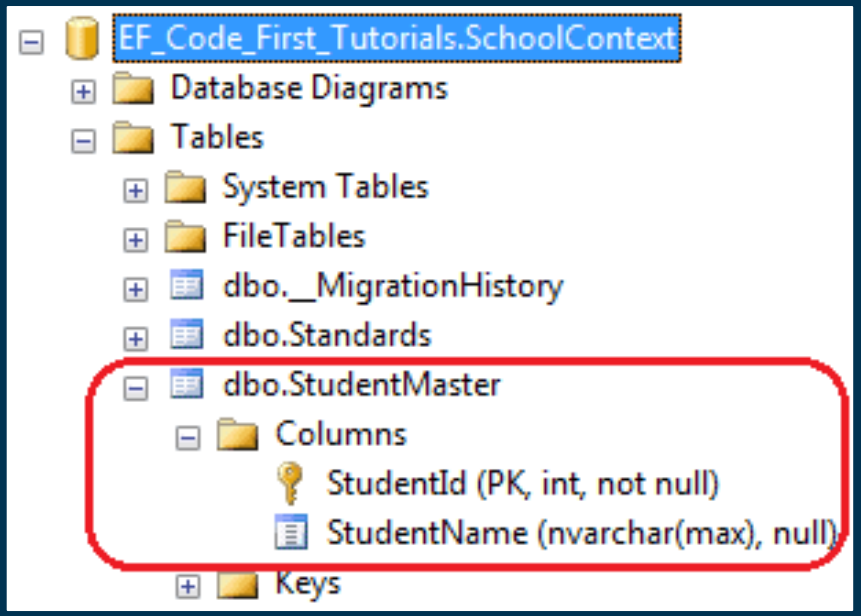
System.ComponentModel.DataAnnotations.Schema Attributes

Attribute	Description
`Table`	Can be applied to an entity class to configure the corresponding table name and schema in the database.
`Column`	Can be applied to a property to configure the corresponding column name, order and data type in the database.
`Index`	Can be applied to a property to configure that the corresponding column should have an Index in the database. (EF 6.1 onwards only)
`ForeignKey`	Can be applied to a property to mark it as a foreign key property.
`NotMapped`	Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database.
`DatabaseGenerated`	Can be applied to a property to configure how the underlying database should generate the value for the corresponding column e.g. identity, computed or none.

Data Annotations - Table Attribute in EF Core

- The `Table` attribute can be applied to a class to configure the corresponding table name in the database.
- It overrides the default convention in EF Core.

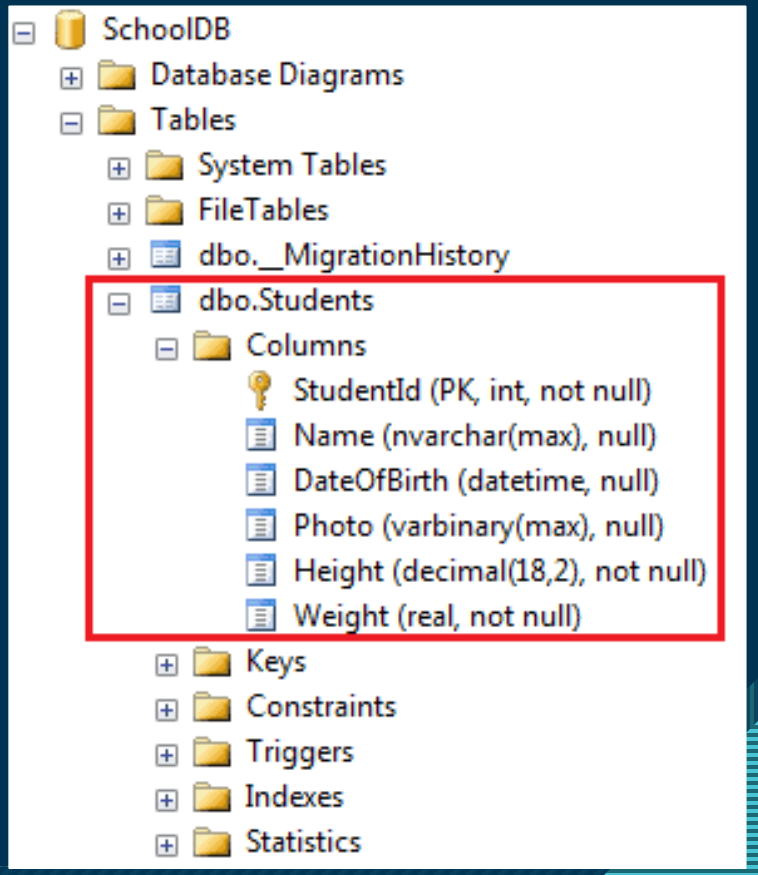
```
using System.ComponentModel.DataAnnotations.Schema;

[Table("StudentMaster")]
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
}
```



Data Annotations - Column Attribute in EF Core

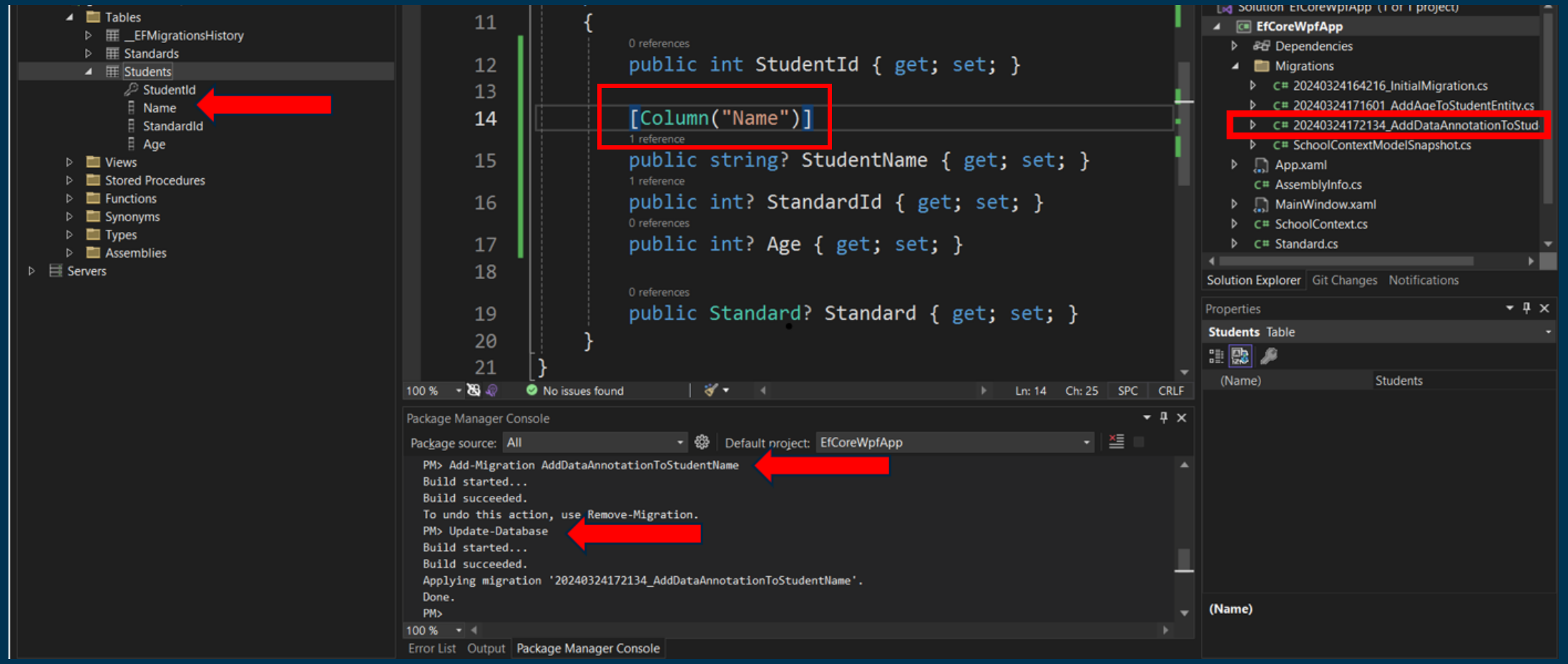
- The `Column` attribute can be applied to one or more properties in an entity class to configure the corresponding column name, data type and order in a database table.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int StudentId { get; set; }

    [Column("Name")]
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
}
```



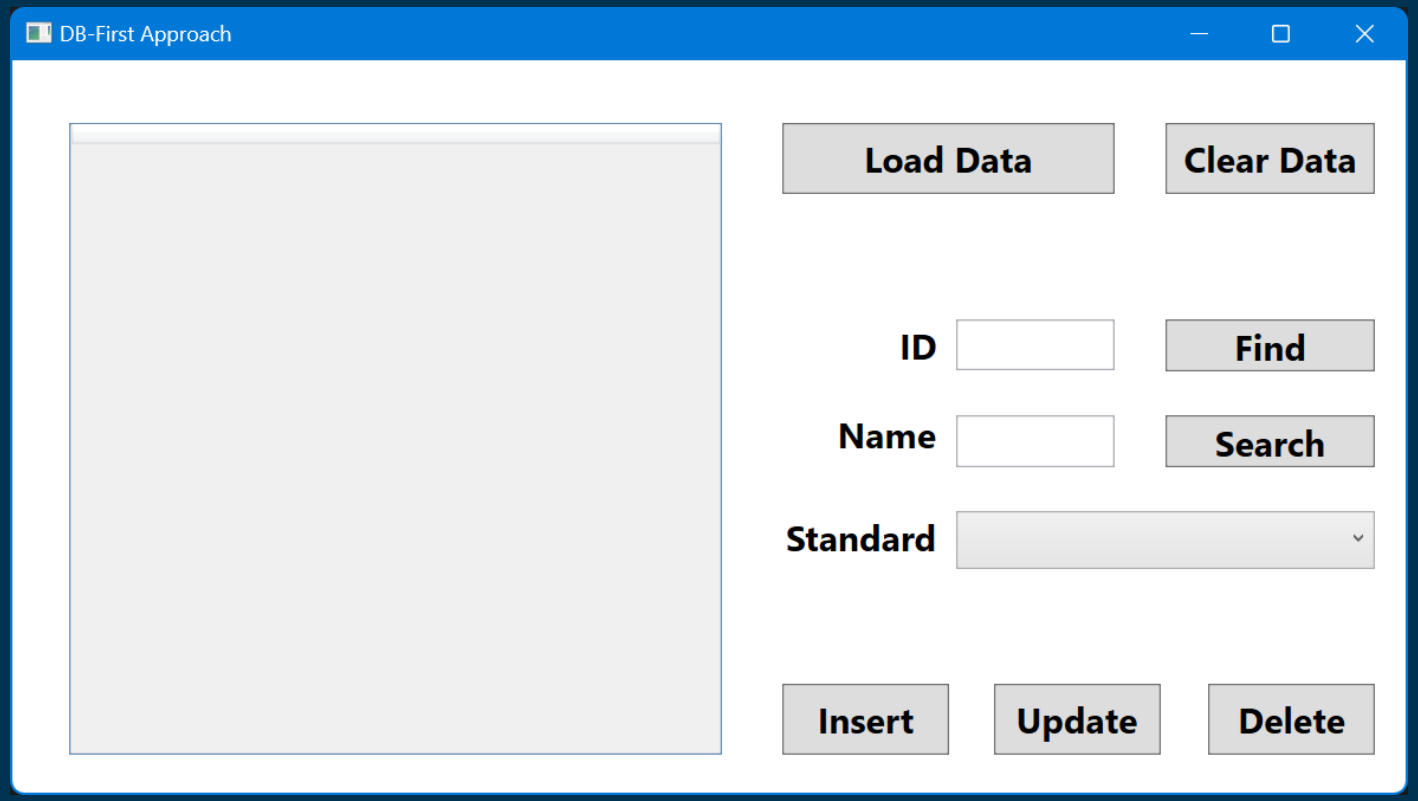
Data Annotations - Column Attribute in EF Core

- Run the migration commands and refresh the database connection.



Exercise

- Complete the CRUD Operations to this application.



Data Annotations Attributes in EF Core

- Learn more about Data Annotations Attributes:
  - Entity Properties:
    - https://learn.microsoft.com/en-us/ef/core/modeling/entity-properties?tabs=data-annotations%2Cwithout-nrt
  - Code First Data Annotations:
    - https://learn.microsoft.com/en-us/ef/ef6/modeling/code-first/data-annotations
  - Entity Framework Tutorial:
    - https://www.entityframeworktutorial.net/code-first/dataannotation-in-code-first.aspx

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Most of the material has been taken as is from:

- Entity Framework Core:
  - https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx