Configuring Nonrelational Properties in EF Core

There are three approaches to configuring Entity Framework Core:

By Convention

Data Annotations

Fluent API

This post is divided into several parts:

EF Core Configuration By Convention

EF Core Configuration via Data Annotations

Using the Fluent API

Excluding Entities or Classes from the Mapping

Primary Key Configuration with Data Annotations and Fluent API

Working with Indexes and Default Values

Recommendations for Using EF Core’s Different Configuration Approaches

EF Core Configuration By Convention

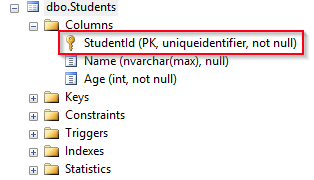
EF Core will follow a set of simple rules

on >>>property types and >>names and configure the database accordingly.

This approach can be overridden by using Data Annotations or Fluent API approach.

property is going to be translated as a primary key in the database

if it has an “Id” property name or a combination <Name of a class> + Id (as it is a case with our StudentId property):



COMPOSITE KEY:

If we have a composite key in our class,

we can’t use the configuration by Convention.

Instead, we have to use either Data Annotations or Fluent API.

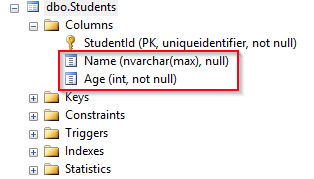
NULLABILITY

When using configuration by convention, the nullability of a column is based on the property type from our model class.

When EF Core uses configuration by convention it will go through all the public properties and map them by their name and type.

Name property can have a Null as a value (because the default value for a string type is null)

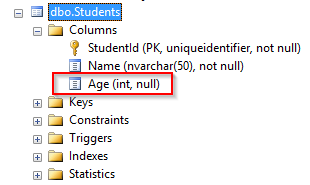
Age cannot (because it is a value type):



if we want the Age property to be nullable in a database,

we can use int? Age

Nullable<int> Age



EF Core Configuration via Data Annotations

use to

validate and

configure the database features.

There are two relevant namespaces

System.ComponentModel.DataAnnotations -property validation

System.ComponentModel.DataAnnotations.Schema. - database configuration

public class Student

{

public Guid StudentId { get; set; }

[Required]

[MaxLength(50, ErrorMessage = "Length must be less then 50 characters")]

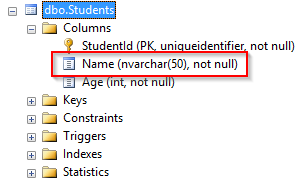
public string Name { get; set; }

public int Age { get; set; }

}

Required attribute, Name field can’t be nullable and

MaxLengh property, limiting the length of that column in a database.



>>>>[Table("Student")]

public class Student

{

>>>>[Column("StudentId")]

public Guid Id { get; set; }

[Required]

[MaxLength(50, ErrorMessage = "Length must be less then 50 characters")]

public string Name { get; set; }

public int? Age { get; set; }

}

Table attribute

[Table] attribute, table to map to in the database WITH DEFAULT SCHEMA

Right now, the name of the table in the database is Students because

the DbSet<T> property is named Students in the ApplicationContext class.

But the [Table] attribute is going to override that.

So, if we need to change a class name,

the [Table] attribute WILL STILL MAP IT TO THE STUDENT TABLE.

If a table, that we are mapping to, belongs to the >>>non-default schema,

we can use the [Table(“TableName”, Schema=”SchemaName”)] attribute,

to provide the information about the required >>>>schema.

Column attribute

The [Column] attribute what column to map to in the database.

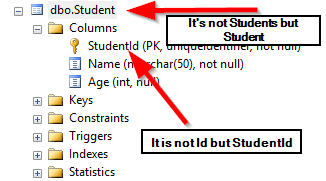
Id property in our class BUT StudentId IN TABLE

[Column] attribute helps us with that.

provide the >>Order and the >>>Database Type of the column with this attribute

[Column(“ColumnName”, Order = 1, TypeName=”nvarchar(50)”)].

After these changes in our class, our table is going to have the same key field but a different name:



Using the Fluent API Approach

The Fluent API is a set of methods that we can use on the ModelBuilder class,

which is available in the OnModelCreating method in our context (ApplicationContext) class.

This approach provides a great variety of the EF Core configuration options

that we can use while configuring our entities.

So, let’s create the OnModelCreating method in the ApplicationContext class

and add the same configuration as we did with the Data Annotations approach:

C#

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Student>()

.ToTable("Student");

modelBuilder.Entity<Student>()

.Property(s => s.Id)

.HasColumnName("StudentId");

modelBuilder.Entity<Student>()

.Property(s => s.Name)

.IsRequired()

.HasMaxLength(50);

modelBuilder.Entity<Student>()

.Property(s => s.Age)

.IsRequired(false);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Student>()

.ToTable("Student");

modelBuilder.Entity<Student>()

.Property(s => s.Id)

.HasColumnName("StudentId");

modelBuilder.Entity<Student>()

.Property(s => s.Name)

.IsRequired()

.HasMaxLength(50);

modelBuilder.Entity<Student>()

.Property(s => s.Age)

.IsRequired(false);

}

In the beginning, we are selecting the entity to configure and with the Property method, we are specifying which property we want to add the constraint on. All the other methods are pretty self-explanatory.

OnModelCreating is called the first time our application instantiates the ApplicationContext class. At that moment all three approaches are applied. As you can see, we haven’t used any method for the primary key, but our table has it nevertheless due to the naming convention:

Recommendations for Using EF Core’s Different Configuration Approaches

By Convention

We should always start with the configuration by Convention.

So, having the same class name as the table name,

having a name for the primary key property that

matches the naming convention and having the properties

with the same name and type as the columns,

should be our first choice.

It is quite easy to prepare this type of configuration and it is time-saving as well.

Data Annotations

For the validation configuration, such as required or max length validation,

you should always prefer the Data Annotations over Fluent API approach. And here’s why:

It is easy to see which validation rule is related to which property because

it is placed right above the property and it is quite self-explanatory

Validations via Data Annotations can be used on the front end because

as we’ve seen in the Student class,

we can configure the error messages if validation fails

We want to use these validation rules prior to the EF Core’s SaveChanges method

(we will talk about it in the following articles).

This approach is going to make our validation code much simpler and easier to maintain

Fluent API

Let’s just say that we should use this approach for everything else.

Of course, we must

use this approach for the configuration that we can’t do otherwise or

when we want to hide the configuration setup from the model class.

So, >>>indexes, >>>composite keys, relationships are the things we should keep in the

OnModelCreating method.

Conclusion

So, we have covered different configuration features that EF Core provides us with.

Of course, there are additional configuration options related to Data Annotations and Fluent API,

and the series is far from over, so we’re going to mention a few of them later on.

In the next part of the series, we are going to learn about

Migrations in EF Core and the Migration features provided by EF Core. So, stay tuned.