**Entity Framework- Model**

**Creating and configuring a model**

Entity Framework uses a set of conventions to build a model based on the shape of your entity classes. You can specify additional configuration to supplement and/or override what was discovered by convention.

## **Use fluent API to configure a model**

You can override the **OnModelCreating** method in your derived context and use the **ModelBuilder** API to configure your model. This is the most powerful method of configuration and allows configuration to be specified without modifying your entity classes. Fluent API configuration has the highest precedence and will override conventions and data annotations.

using Microsoft.EntityFrameworkCore;

namespace EFModeling.FluentAPI.Required

{

class MyContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

#region Required

protected override void OnModelCreating(ModelBuilder modelBuilder)

{ modelBuilder.Entity<Blog>() .Property(b => b.Url)

.IsRequired();

}

#endregion

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

}

}

Alternately, use Data Annotations. Data annotations will override conventions, but will be overridden by Fluent API configuration.

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

namespace EFModeling.DataAnnotations.Required

{

class MyContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

}

#region Required

public class Blog

{

public int BlogId { get; set; }

[Required]

public string Url { get; set; }

}

#endregion

}

# Entity Types

Including a DbSet of a type on your context means that it is included in EF Core's model; we usually refer to such a type as an entity. EF Core can read and write entity instances from/to the database, and if you're using a relational database, EF Core can create tables for your entities via migrations.

## **Including types in the model**

By convention, types that are exposed in DbSet properties on your context are included in the model as entities. Entity types that are specified in the OnModelCreating method are also included, as are any types that are found by recursively exploring the navigation properties of other discovered entity types.

In the code sample below, all types are included:

* Blog is included because it's exposed in a DbSet property on the context.
* Post is included because it's discovered via the Blog.Posts navigation property.
* AuditEntry because it is specified in OnModelCreating.

class MyContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<AuditEntry>();

}

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public List<Post> Posts { get; set; }

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public Blog Blog { get; set; }

}

public class AuditEntry

{

public int AuditEntryId { get; set; }

public string Username { get; set; }

public string Action { get; set; }

}

## Excluding types from the model

[NotMapped]

public class BlogMetadata

{

public DateTime LoadedFromDatabase { get; set; }

}

## **Table name**

By convention, each entity type will be set up to map to a database table with the same name as the DbSet property that exposes the entity. **If no DbSet exists for the given entity, the class name is used.**

You can manually configure the table name:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>() .ToTable("blogs", schema: "blogging");

}

Or

[Table("blogs", Schema = "blogging")]

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

}

Rather than specifying the schema for each table, you can also define the default schema at the model level with the fluent API:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.HasDefaultSchema("blogging");

}

Note that setting the default schema will also affect other database objects, such as sequences.

# Entity Properties

Each entity type in your model has a set of properties, which EF Core will read and write from the database. If you're using a relational database, entity properties map to table columns.

## **Included and excluded properties**

By convention, all **public properties with a getter and a setter** will be included in the model.

Specific properties **can be excluded as** follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Ignore(b => b.LoadedFromDatabase);

}

Or

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

[NotMapped]

public DateTime LoadedFromDatabase { get; set; }

}

## **Column names**

By convention, when using a relational database, entity properties are mapped to table columns having the same name as the property.

If you prefer to configure your columns with different names, you can do so as following:

public class Blog

{

[Column("blog\_id")]

public int BlogId { get; set; }

public string Url { get; set; }

}

Or

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>() .Property(b => b.BlogId) .HasColumnName("blog\_id");

}

## **Column data types**

When using a relational database, the database provider selects a data type based on the .NET type of the property. It also takes into account other metadata, such as the configured [maximum length](https://docs.microsoft.com/en-us/ef/core/modeling/entity-properties?tabs=fluent-api%2Cwithout-nrt#maximum-length), whether the property is part of a primary key, etc.

For example, SQL Server maps DateTime properties to datetime2(7) columns, and string properties to nvarchar(max) columns (or to nvarchar(450) for properties that are used as a key).

You can also configure your columns to specify an exact data type for a column. For example the following code configures Url as a non-unicode string with maximum length of 200 and Rating as decimal with precision of 5 and scale of 2:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>(eb =>

{

eb.Property(b => b.Url).HasColumnType("varchar(200)"); eb.Property(b => b.Rating).HasColumnType("decimal(5, 2)");

});

}

Or

public class Blog

{

public int BlogId { get; set; }

[Column(TypeName = "varchar(200)")]

public string Url { get; set; }

[Column(TypeName = "decimal(5, 2)")]

public decimal Rating { get; set; }

}

### **Maximum length**

Configuring a maximum length provides a hint to the database provider about the appropriate column data type to choose for a given property. Maximum length only applies to array data types, such as string and byte[].

Entity Framework does not do any validation of maximum length before passing data to the provider. It is up to the provider or data store to validate if appropriate. For example, when targeting SQL Server, exceeding the maximum length will result in an exception as the data type of the underlying column will not allow excess data to be stored.

In the following example, configuring a maximum length of 500 will cause a column of type nvarchar(500) to be created on SQL Server:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>() .Property(b => b.Url) .HasMaxLength(500);

}

OR

public class Blog

{

public int BlogId { get; set; }

[MaxLength(500)]

public string Url { get; set; }

}

## **Required and optional properties**

A property is considered optional if it is valid for it to contain null. If null is not a valid value to be assigned to a property then it is considered to be a required property. When mapping to a relational database schema, required properties are created as non-nullable columns, and optional properties are created as nullable columns.

### Conventions

By convention, a property whose .NET type can contain null will be configured as optional, whereas properties whose .NET type cannot contain null will be configured as required. For example, all properties with .NET value types (int, decimal, bool, etc.) are configured as required, and all properties with nullable .NET value types (int?, decimal?, bool?, etc.) are configured as optional.

C# 8 introduced a new feature called [nullable reference types](https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/nullable-reference-types), which allows reference types to be annotated, indicating whether it is valid for them to contain null or not. This feature is disabled by default, and if enabled, it modifies EF Core's behavior in the following way:

* If nullable reference types are disabled (the default), all properties with .NET reference types are configured as optional by convention (e.g. string).
* If nullable reference types are enabled, properties will be configured based on the C# nullability of their .NET type: string? will be configured as optional, whereas string will be configured as required.

The following example shows an entity type with required and optional properties, with the nullable reference feature disabled (the default) and enabled:

#nullable disable

public class CustomerWithoutNullableReferenceTypes

{

public int Id { get; set; }

[Required] // Data annotations needed to configure as required

public string FirstName { get; set; }

[Required]

public string LastName { get; set; } // Data annotations needed to configure as required public string MiddleName { get; set; } // Optional by convention

}

#nullable enable

public class Customer

{

public int Id { get; set; }

public string FirstName { get; set; } // Required by convention

public string LastName { get; set; } // Required by convention

public string? MiddleName { get; set; } // Optional by convention

public Customer(string firstName, string lastName, string? middleName = null)

{

FirstName = firstName;

LastName = lastName;

MiddleName = middleName;

}

}

Using nullable reference types is recommended since it flows the nullability expressed in C# code to EF Core's model and to the database, and obviates the use of the Fluent API or Data Annotations to express the same concept twice.

# Keys

A key serves as a unique identifier for each entity instance. Most entities in EF have a single key, which maps to the concept of a primary key in relational databases. (for entities without keys, see [Keyless entities](https://docs.microsoft.com/en-us/ef/core/modeling/keyless-entity-types)

By convention, a property named Id or <type name>Id will be configured as the primary key of an entity.

class Car

{

**public string Id { get; set; }**

public string Make { get; set; }

public string Model { get; set; }

}

class Truck

{

**public string TruckId { get; set; }**

public string Make { get; set; }

public string Model { get; set; }

}

 You can configure a single property to be the primary key of an entity as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Car>()

.HasKey(c => c.LicensePlate);

}

Or

class Car

{

[Key]

public string LicensePlate { get; set; }

public string Make { get; set; }

public string Model { get; set; }

}

You can also configure multiple properties to be the key of an entity - this is known as a composite key**. Composite keys** can only be configured using the Fluent API; conventions will never setup a composite key, and you can not use Data Annotations to configure one.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Car>()

.HasKey(c => new { c.State, c.LicensePlate });

}

## **Primary key name**

By convention, on relational databases primary keys are created with the name PK\_<type name>. You can configure the name of the primary key constraint as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasKey(b => b.BlogId)

.HasName("PrimaryKey\_BlogId");

}

## **Key types and values**

While EF Core supports using properties of any primitive type as the primary key, including string, Guid, byte[] and others, not all databases support all types as keys. In some cases the key values can be converted to a supported type automatically, otherwise the conversion should be [specified manually](https://docs.microsoft.com/en-us/ef/core/modeling/value-conversions).

Key properties must always have a non-default value when adding a new entity to the context, but some types will be [generated by the database](https://docs.microsoft.com/en-us/ef/core/modeling/generated-properties). In that case EF will try to generate a temporary value when the entity is added for tracking purposes. After [SaveChanges](https://docs.microsoft.com/en-us/dotnet/api/Microsoft.EntityFrameworkCore.DbContext.SaveChanges) is called the temporary value will be replaced by the value generated by the database.

If a key property has its value generated by the database and a non-default value is specified when an entity is added, then EF will assume that the entity already exists in the database and will try to update it instead of inserting a new one. To avoid this turn off value generation or see [**how to specify explicit values for generated properties**](https://docs.microsoft.com/en-us/ef/core/saving/explicit-values-generated-properties).

## **Alternate Keys**

An alternate key serves as an alternate unique identifier for each entity instance in addition to the primary key; it can be used as the target of a relationship. When using a relational database this maps to the concept of a unique index/constraint on the alternate key column(s) and one or more foreign key constraints that reference the column(s).

If you just want to enforce uniqueness on a column, define a unique index rather than an alternate key (see [**Indexes**](https://docs.microsoft.com/en-us/ef/core/modeling/indexes)). In EF, alternate keys are read-only and provide additional semantics over unique indexes because they can be used as the target of a foreign key.

Alternate keys are typically introduced for you when needed and you do not need to manually configure them. By convention, an alternate key is introduced for you when you identify a property which isn't the primary key as the target of a relationship.

<https://docs.microsoft.com/en-us/ef/core/modeling/keys?tabs=data-annotations>