Entity Framework Fluent API - Configuring and Mapping Properties and Types

Updated: October 23, 2016

When working with Entity Framework Code First the default behavior is to map your POCO classes to tables using a set of conventions baked into EF. Sometimes, however, you cannot or do not want to follow those conventions and need to map entities to something other than what the conventions dictate.

There are two main ways you can configure EF to use something other than conventions, namely annotations or EFs fluent API. The annotations only cover a subse of the fluent API functionality, so there are mapping scenarios that cannot be achieved using annotations. This article is designed to demonstrate how to use the fluent API to configure properties.

The code first fluent API is most commonly accessed by overriding the OnModelCreating method on your derived DbContext. The following samples are designed to show how to do various tasks with the fluent api and allow you to copy the code out and customize it to suit your model, if you wish to see the model that they can be used with as-is then it is provided at the end of this article.

Model-Wide Settings

Default Schema (EF6 onwards)

Starting with EF6 you can use the HasDefaultSchema method on DbModelBuilder to specify the database schema to use for all tables, stored procedures, etc. This default setting will be overridden for any objects that you explicitly configure a different schema for.

```
modelBuilder.HasDefaultSchema("sales");
```

Custom Conventions (EF6 onwards)

Starting with EF6 you can create your own conventions to supplement the ones included in Code First. For more details, see Custom Code First Conventions.

Property Mapping

The Property method is used to configure attributes for each property belonging to an entity or complex type. The Property method is used to obtain a configuration object for a given property. The options on the configuration object are specific to the type being configured; IsUnicode is available only on string properties for example.

Configuring a Primary Key

The Entity Framework convention for primary keys is:

1. Your class defines a property whose name is "ID" or "Id"
2. or a class name followed by "ID" or "Id"

To explicitly set a property to be a primary key, you can use the HasKey method. In the following example, the HasKey method is used to configure the InstructorID primary key on the OfficeAssignment type.

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

Configuring a Composite Primary Key

The following example configures the DepartmentID and Name properties to be the composite primary key of the Department type.

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

Switching off Identity for Numeric Primary Keys

The following example sets the DepartmentID property to System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.None to indicate that the value will not be generated by the database.

```
modelBuilder.Entity<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

Specifying the Maximum Length on a Property

In the following example, the Name property should be no longer than 50 characters. If you make the value longer than 50 characters, you will get a DbEntityValidationException exception. If Code First creates a database from this model it will also set the maximum length of the Name column to 50 characters.

```
modelBuilder.Entity<Department>().Property(t => t.Name).HasMaxLength(50);
```

Configuring the Property to be Required

In the following example, the Name property is required. If you do not specify the Name, you will get a DbEntityValidationException exception. If Code First creates a database from this model then the column used to store this property will usually be non-nullable.

Note: In some cases it may not be possible for the column in the database to be non-nullable even though the property is required. For example, when using a TPH inheritance strategy data for multiple types is stored in a single table. If a derived type includes a required property the column cannot be made non-nullable since not all types in the hierarchy will have this property.

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

Configuring an Index on one or more properties

EF6.1 Onwards Only - The Index attribute was introduced in Entity Framework 6.1. If you are using an earlier version the information in this section does not apply

Creating indexes isn't natively supported by the Fluent API, but you can make use of the support for **IndexAttribute** via the Fluent API. Index attributes are processed by including a model annotation on the model that is then turned into an Index in the database later in the pipeline. You can manually add these same annotations using the Fluent API.

The easiest way to do this is to create an instance of **IndexAttribute** that contains all the settings for the new index. You can then create an instance of **IndexAnnotation** which is an EF specific type that will convert the **IndexAttribute** settings into a model annotation that can be stored on the EF model. These can then be passed to the **HasColumnAnnotation** method on the Fluent API, specifying the name **Index** for the annotation.

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute()));
```

For a complete list of the settings available in **IndexAttribute**, see the *Index* section of Code First Data Annotations. This includes customizing the index name, creating unique indexes, and creating multi-column indexes.

You can specify multiple index annotations on a single property by passing an array of **IndexAttribute** to the constructor of **IndexAnnotation**.

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation(
        "Index",
        new IndexAnnotation(new[]
            {
                new IndexAttribute("Index1"),
                new IndexAttribute("Index2") { IsUnique = true }
            })));
```

Specifying Not to Map a CLR Property to a Column in the Database

The following example shows how to specify that a property on a CLR type is not mapped to a column in the database.

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

Mapping a CLR Property to a Specific Column in the Database

The following example maps the Name CLR property to the DepartmentName database column.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

Renaming a Foreign Key That Is Not Defined in the Model

If you choose not to define a foreign key on a CLR type, but want to specify what name it should have in the database, do the following:

```
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Configuring whether a String Property Supports Unicode Content

By default strings are Unicode (nvarchar in SQL Server). You can use the IsUnicode method to specify that a string should be of varchar type.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .IsUnicode(false);
```

Configuring the Data Type of a Database Column

The HasColumnType method enables mapping to different representations of the same basic type. Using this method does not enable you to perform any conversion of the data at run time. Note that IsUnicode is the preferred way of setting columns to varchar, as it is database agnostic.

```
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

Configuring Properties on a Complex Type

There are two ways to configure scalar properties on a complex type.

You can call Property on ComplexTypeConfiguration.

```
modelBuilder.ComplexType<Details>()
    .Property(t => t.Location)
    .HasMaxLength(20);
```

You can also use the dot notation to access a property of a complex type.

```
modelBuilder.Entity<OnsiteCourse>()
    .Property(t => t.Details.Location)
    .HasMaxLength(20);
```

Configuring a Property to Be Used as an Optimistic Concurrency Token

To specify that a property in an entity represents a concurrency token, you can use either the ConcurrencyCheck attribute or the IsConcurrencyToken method.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsConcurrencyToken();
```

You can also use the IsRowVersion method to configure the property to be a row version in the database. Setting the property to be a row version automatically configures it to be an optimistic concurrency token.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsRowVersion();
```

Type Mapping

Specifying That a Class Is a Complex Type

By convention, a type that has no primary key specified is treated as a complex type. There are some scenarios where Code First will not detect a complex type (for example, if you do have a property called ID, but you do not mean for it to be a primary key). In such cases, you would use the fluent API to explicitly specify that a type is a complex type.

```
modelBuilder.ComplexType<Details>();
```

Specifying Not to Map a CLR Entity Type to a Table in the Database

The following example shows how to exclude a CLR type from being mapped to a table in the database.

```
modelBuilder.Ignore<OnlineCourse>();
```

Mapping an Entity Type to a Specific Table in the Database

All properties of Department will be mapped to columns in a table called t_ Department.

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

You can also specify the schema name like this:

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department", "school");
```

Mapping the Table-Per-Hierarchy (TPH) Inheritance

In the TPH mapping scenario, all types in an inheritance hierarchy are mapped to a single table. A discriminator column is used to identify the type of each row. When creating your model with Code First, TPH is the default strategy for the types that participate in the inheritance hierarchy. By default, the discriminator column is added to the table with the name "Discriminator" and the CLR type name of each type in the hierarchy is used for the discriminator values. You can modify the default behavior by using the fluent API.

```
modelBuilder.Entity<Course>()
    .Map<Course>(m => m.Requires("Type").HasValue("Course"))
    .Map<OnsiteCourse>(m => m.Requires("Type").HasValue("OnsiteCourse"));
```

Mapping the Table-Per-Type (TPT) Inheritance

In the TPT mapping scenario, all types are mapped to individual tables. Properties that belong solely to a base type or derived type are stored in a table that maps to that type. Tables that map to derived types also store a foreign key that joins the derived table with the base table.

```
modelBuilder.Entity<Course>().ToTable("Course");
modelBuilder.Entity<OnsiteCourse>().ToTable("OnsiteCourse");
```

Mapping the Table-Per-Concrete Class (TPC) Inheritance

In the TPC mapping scenario, all non-abstract types in the hierarchy are mapped to individual tables. The tables that map to the derived classes have no relationship to the table that maps to the base class in the database. All properties of a class, including inherited properties, are mapped to columns of the corresponding table.

Call the MapInheritedProperties method to configure each derived type. MapInheritedProperties remaps all properties that were inherited from the base class to new columns in the table for the derived class.

Note: Note that because the tables participating in TPC inheritance hierarchy do not share a primary key there will be duplicate entity keys when inserting in tables that are mapped to subclasses if you have database generated values with the same identity seed. To solve this problem you can either specify a different initial seed value for each table or switch off identity on the primary key property. Identity is the default value for integer key properties when working with Code First.

```
modelBuilder.Entity<Course>()
    .Property(c => c.CourseID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

modelBuilder.Entity<OnsiteCourse>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("OnsiteCourse");
});

modelBuilder.Entity<OnlineCourse>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("OnlineCourse");
});
```

Mapping Properties of an Entity Type to Multiple Tables in the Database (Entity Splitting)

Entity splitting allows the properties of an entity type to be spread across multiple tables. In the following example, the Department entity is split into two tables: Department and DepartmentDetails. Entity splitting uses multiple calls to the Map method to map a subset of properties to a specific table.

```
modelBuilder.Entity<Department>()
    .Map(m =>
    {
        m.Properties(t => new { t.DepartmentID, t.Name });
        m.ToTable("Department");
    })
    .Map(m =>
    {
        m.Properties(t => new { t.DepartmentID, t.Administrator, t.StartDate, t.Budget });
        m.ToTable("DepartmentDetails");
    });
```

Mapping Multiple Entity Types to One Table in the Database (Table Splitting)

The following example maps two entity types that share a primary key to one table.

```
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);
```

```
modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");
```

Mapping an Entity Type to Insert/Update/Delete Stored Procedures (EF6 onwards)

Starting with EF6 you can map an entity to use stored procedures for insert update and delete. For more details see, Code First Insert/Update/Delete Stored Procedures.

Model Used in Samples

The following Code First model is used for the samples on this page.

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
// add a reference to System.ComponentModel.DataAnnotations DLL
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System;

public class SchoolEntities : DbContext
{
    public DbSet<Course> Courses { get; set; }
    public DbSet<Department> Departments { get; set; }
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention then the generated tables will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}

public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public System.DateTime StartDate { get; set; }
    public int? Administrator { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; private set; }
}

public class Course
{
    public Course()
    {
        this.Instructors = new HashSet<Instructor>();
    }
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
    public virtual ICollection<Instructor> Instructors { get; private set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}
```

```
public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}

public class Instructor
{
    public Instructor()
    {
        this.Courses = new List<Course>();
    }

    // Primary key
    public int InstructorID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public System.DateTime HireDate { get; set; }

    // Navigation properties
    public virtual ICollection<Course> Courses { get; private set; }
}

public class OfficeAssignment
{
    // Specifying InstructorID as a primary
    [Key()]
    public Int32 InstructorID { get; set; }

    public string Location { get; set; }

    // When the Entity Framework sees Timestamp attribute
    // it configures ConcurrencyCheck and DatabaseGeneratedPattern=Computed.
    [Timestamp]
    public Byte[] Timestamp { get; set; }

    // Navigation property
    public virtual Instructor Instructor { get; set; }
}
```